



Concurrent Object-Oriented Programming

Bertrand Meyer, Piotr Nienaltowski



Lecture 12-13:

Advanced OO techniques in SCOOP



Outline

3

- Once functions

- Inheritance
 - Polymorphism
 - Deferred classes
 - Feature redefinition
 - Precursor calls

- Genericity
 - Constrained genericity

- Agents



Once functions

4

```
barrier: separate MY_BARRIER  
  -- Barrier.  
  once  
  create Result.make (3)  
  end
```

- Similar to constants
 - Always return the same value
- Lazy evaluation
 - Body executed on first access
 - Once-per-class semantics
- Examples of use
 - Heavy computations
 - Stock market statistics
 - Common contact point for objects of one type
 - Feature *io* in class *ANY*
 - Queue for meeting Santa



Once functions in concurrent context

5

- Is once-per-system semantics always correct?

```
barrier: separate MY_BARRIER
```

```
-- 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**



Scoop2scoopli and once functions

6

- Preprocessor applies the appropriate semantics

```
barrier: separate MY_BARRIER is  
once  
    . . .  
end
```

becomes

```
barrier: SCOOP_SEPARATE__MY_BARRIER is  
indexing  
    once_status: global  
once  
    . . .  
end
```



Inheritance

7

- Polymorphism
- Deferred classes
- Feature redefinition
- Precursor calls



Polymorphism

```
class ELF inherit SANTA_HELPER ... end
```

```
class REINDEER inherit SANTA_HELPER ... end
```

```
class SANTA
```

```
...  
feature
```

```
  elf: separate ELF
```

```
  reindeer: separate REINDEER
```

```
...  
  meet_helper (a_helper: separate SANTA_HELPER) is
```

```
    require
```

```
      a_helper.is_ready
```

```
    do
```

```
      a_helper.talk_it_over (Current)
```

```
    end
```

```
...  
  meet_helper (elf)
```

```
-- Polymorphic attachment
```

```
  meet_helper (reindeer)
```

```
-- Polymorphic attachment
```

```
...  
end
```



Scoop2scoopli and polymorphism

9

```
class SANTA
```

```
...
```

```
feature
```

```
elf: separate ELF
```

```
local_elf: ELF
```

```
...
```

```
meet_helper (a_helper: separate SANTA_HELPER) is
```

```
  require
```

```
    a_helper.is_ready
```

```
  do
```

```
    a_helper.talk_it_over (Current)
```

```
  end
```

```
...
```

```
meet_helper (elf)
```

```
meet_helper (local_elf)
```

```
...
```

```
end
```

-- This works.

-- This doesn't because it requires
-- both polymorphism and conversion.
-- Unfortunately, they do not combine.



Solution

- First solution: conversion followed by polymorphism

aux_elf: **separate** ELF

local_elf: ELF

...

aux_elf := *local_elf*

-- Conversion.

meet_helper (*aux_elf*)

-- Polymorphism.

- Second solution: polymorphism followed by conversion

aux_santa_helper: SANTA_HELPER

local_elf: ELF

...

aux_santa_helper := *local_elf*

-- Polymorphism.

meet_helper (*aux_santa_helper*)

-- Conversion.

- Second solution only works if *SANTA_HELPER* is not **deferred**



Deferred classes

11

deferred class *SANTA_HELPER* . . . **End**

- Conversion cannot be used on deferred types

local_santa_helper: SANTA_HELPER

*santa_helper: **separate** SANTA_HELPER*

. . .

santa_helper := local_santa_helper **-- Invalid.**

- Solution: use non-deferred subtype of *SANTA_HELPER*



Feature redefinition

12

- Usual rules of DbC apply
- Preconditions may be kept or weakened
- Postconditions and invariants may be kept or strengthened
- Result types may be redefined covariantly
- Formal argument types may be redefined
 - Covariantly w.r.t. class type
 - Contravariantly w.r.t. attached tag and processor tag



Inheritance: preconditions

13

- Preconditions may be kept or weakened
 - *Less waiting*
 - Full support for preconditions in `scoop2scoopli`

-- ancestor

```
r (x: separate X )  
  require  
    x.count > 10  
  do  
    . . .  
  end  
r (my_x)
```

-- descendant

```
r (x: separate X )  
  require else  
    x.count > 0  
  do  
    . . .  
  end
```

- Contracts are bound *dynamically*
 - demo



Inheritance: postconditions and invariants

14

- Postconditions may be kept or strengthened
 - More guarantees to the client
 - **Scoop2scoopli applies sequential semantics**
 - Postcondition checking **turned off** by default
- Invariants may be kept or strengthened
 - Full support for invariant checking in scoop2scoopli
 - Invariant checking **turned on** by default
- **See "Contracts for concurrency" paper on the website**



Inheritance: postconditions and invariants

15

- Result types may be redefined covariantly

```
-- ancestor  
my_x: separate X
```

```
-- descendant  
my_x: X
```

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

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

```
r (my_x)  
-- Blocking call
```

```
-- Ancestor's r (my_x) is now a  
-- non-blocking call.  
-- Is it a problem?
```

- No problem here: client waits less



Argument types: attached tags

- Formal argument types may be redefined
 - Covariantly w.r.t. class type
 - Contravariantly w.r.t. **attached tag** and processor tag

-- ancestor

my_y: *Y*

r (*y*: **separate** *Y*)

do

....

end

r (*my_y*)

-- Blocking call

-- descendant

my_y: *Y*

r (*y*: ? **separate** *Y*)

do

....

end

-- Ancestor's *r* (*my_y*) is now a

-- non-blocking call.

-- Is it a problem?

- No problem here: client waits less



Argument types: processor tags

17

- Formal argument types may be redefined
 - Covariantly w.r.t. class type
 - Contravariantly w.r.t. attached tag and **processor tag**

-- ancestor

my_y: Y

r (y: Y)

do

....

end

r (my_y)

-- Non-blocking call

-- descendant

my_y: Y

*r (y: **separate** Y)*

do

....

end

-- Ancestor's *r (my_y)* is now a

-- blocking call. Is it a problem?

- No problem here: client may only use non-separate actual
 - Why?
 - Is *r (my_y)* really blocking?



Precursor calls

18

- Precursor call invokes ancestor's version of redefined routine.
- Allows reuse of routine implementation.

```
class ELF
```

```
inherit SANTA_HELPER redefine make end
```

```
feature -- Initialization
```

```
  make (an_id: INTEGER)
```

```
    require
```

```
      an_id > 0
```

```
    do
```

```
      Precursor {SANTA_HELPER} (an_id)
```

```
      -- Additional stuff here.
```

```
    end
```

```
    . . .
```

```
end
```



Scoop2scoopli and precursor calls

19

- Scoop2scoopli uses renaming to implement dynamic binding of routine bodies and wait-conditions
- Therefore, explicit ancestor annotations are required

Precursor {*SANTA_HELPER*} (*an_id*) -- Valid.

Precursor (*an_id*) -- Invalid.

- This only applies to routines that take separate arguments
- It is a good habit to indicate the ancestor anyway.



Scoop2scoopli: feature redefinition

20

- Preconditions: **fully supported**
- Postconditions: **supported** but sequential semantics applies
- Invariants: **fully supported**
- Redefinition of class types: **fully supported**
- Redefinition of processor tags and detachable tags: **not supported.**
- Precursor calls: **supported** but explicit ancestor name required
- See "Flexible locking in SCOOP" paper on the website



- Entities of generic types may be separate

list: LIST [X]

*list: **separate** LIST [X]*

- Actual generic parameters may be separate

*list: LIST [**separate** X]*

*list: **separate** LIST [**separate** X]*

- All combinations are meaningful and useful



Genericity: even more fun

22

- Actual generic parameters may be also of generic type

list: **LIST** [**separate** ARRAY [**SET** [**separate** X]]]

- Separateness is relative to object of generic class, e.g. elements of

list: **separate** LIST [X]

are non-separate w.r.t. *list* but separate w.r.t. **Current**.

- Type combinators apply



Scoop2scoopli and genericity

23

- Full support

list: LIST [X]

list: LIST [b>separate X]

list: b>separate LIST [X] -- X expanded

list: b>separate LIST [b>separate X]

- No support

list: b>separate LIST [X] -- X non-expanded

But it can be simulated with

list: b>separate LIST [b>separate X]



Constrained genericity

24

```
class PRIORITY_QUEUE [G -> PART_COMPARABLE]
```

- Actual generic parameter must conform
queue: **separate** *PRIORITY_QUEUE* [X]
-- X must inherit from *PART_COMPARABLE*

- *Scoop2scoopli* allows

```
queue: PRIORITY_QUEUE [X]  
queue: separate PRIORITY_QUEUE [X]
```

but disallows

```
queue: PRIORITY_QUEUE [separate X]  
queue: separate PRIORITY_QUEUE [separate X]
```



Constrained genericity

25

- Try

class *MY_QUEUE* [G -> **separate** *SANTA_HELPER*]

q: **separate** *MY_QUEUE* [**separate** *REINDEER*]

- Scoop2scoopli accepts it
- Precompiled code compiles correctly
- **ES syntax checker complains**
 - Just ignore it



- What are agents?
- Do we need special rules to handle them in SCOOP?
 - We hate special rules, don't we?
- What agents can do for us
 - Convenience
 - Full asynchrony
 - Parallel wait
- Demo



What are agents?

27

- An agent represents an operation ready to be called

$x: X$

$my_operation: ROUTINE [X, TUPLE]$

$my_operation := \mathbf{agent} \ x.f$

...

$my_operation.call ([])$ -- Just like calling $x.f$

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

$y.r (my_operation)$ -- y will call agent later on



What are agents?

28

- Arguments can be **closed** (fixed) or **open**

```
my_operation := agent io.put_string ("Hello World!")  
my_operation.call ([]) -- no arguments necessary;  
                        -- use empty tuple
```

```
my_operation := agent io.put_string (?)  
my_operation.call (["Hello World!"]) -- 1 argument
```

- 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

29

- Object-oriented wrappers for operations
 - > strongly-typed function pointers (C++)
 - ~ .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
 - Model - View - Controller

- Replace several patterns
 - Observer
 - Visitor
 - ...



Problematic agents

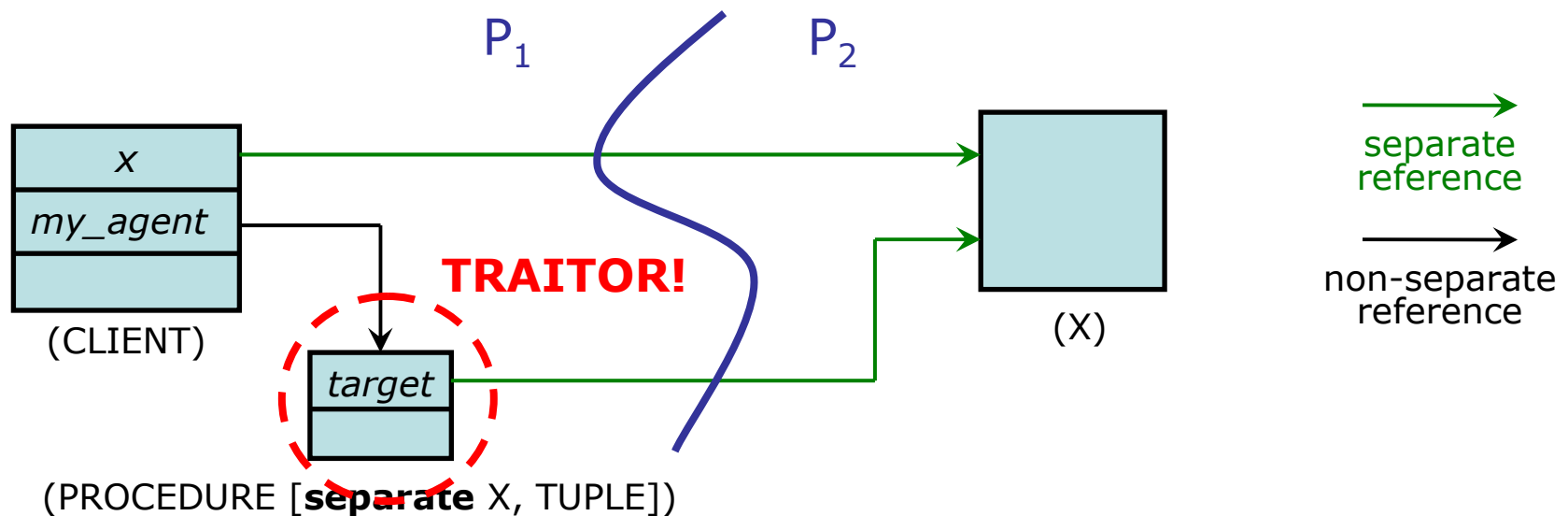
```
my_agent: PROCEDURE [separate ANY, TUPLE]
```

```
x: separate X
```

```
...
```

```
my_agent := agent x.f
```

```
my_agent.call ([])      -- Like x.f without locking x!
```

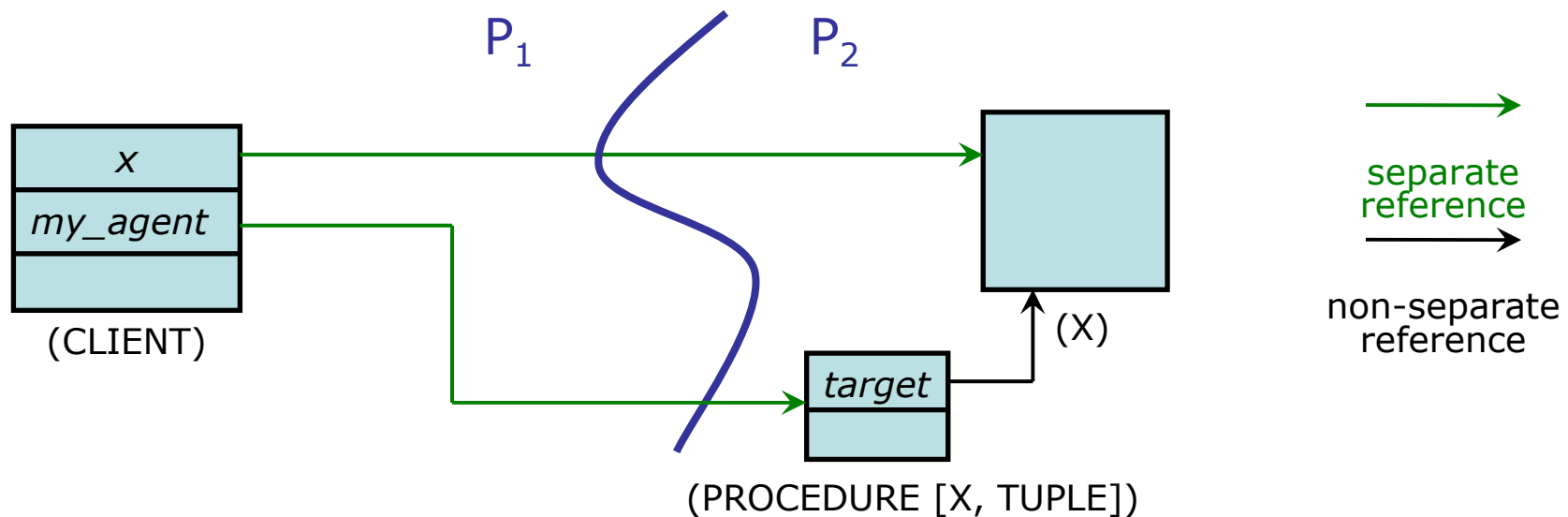




Let's make the agent separate!

31

```
my_agent: separate PROCEDURE [X, TUPLE]
x: separate X
...
my_agent := agent x.f
-- agent x.f handled by x's processor
my_agent.call ([])      -- Invalid call!
```





Separate agents

32

- Agent built on a separate call becomes itself *separate*
 - It is handled by the same processor as its target

```
my_agent: separate PROCEDURE [X, TUPLE]
```

```
x: separate X
```

```
...
```

```
my_agent := agent x.f
```

```
call (my_agent)
```

```
call (an_agent: separate PROCEDURE [ANY, TUPLE])
```

```
  do
```

```
    an_agent.call ([])  -- Valid separate call
```

```
  end
```

- No special rules for separate agents +
- Agents pass processors' boundaries just as other objects do +



1st benefit: convenience

33

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

```
my_x: separate X
```

```
r (my_x)      -- x.f
```

```
s (my_x)      -- x.g (5, "Hello")
```

```
...
```

```
r (x: separate X)
```

```
  do
```

```
    x.f
```

```
  end
```

```
s (x: separate X)
```

```
  do
```

```
    x.g (5, "Hello")
```

```
  end
```

- With agents, we can write universal enclosing routine

```
call (agent my_x.f)
```

```
call (agent my_x.g (5, "Hello"))
```

```
call (an_agent: separate PROCEDURE [ANY, TUPLE])
```

```
  -- Universal enclosing routine.
```

```
  do
```

```
    an_agent.call ([])
```

```
  en
```



2nd benefit: full asynchrony

34

- Without agents, full asynchrony cannot be achieved

*my_x, my_y: **separate** X*

...

r (my_x) -- Blocking call

do_local_stuff

...

*r (x: **separate** X)*

do

x.f -- Asynchronous

end

- With agents, it's easy-peasy

*asynch (**agent** my_x.f) -- Non-blocking call*

do_local_stuff

...

*asynch (an_agent: ?**separate** PROCEDURE [ANY, TUPLE])*

-- Call 'an_agent' asynchronously.

do

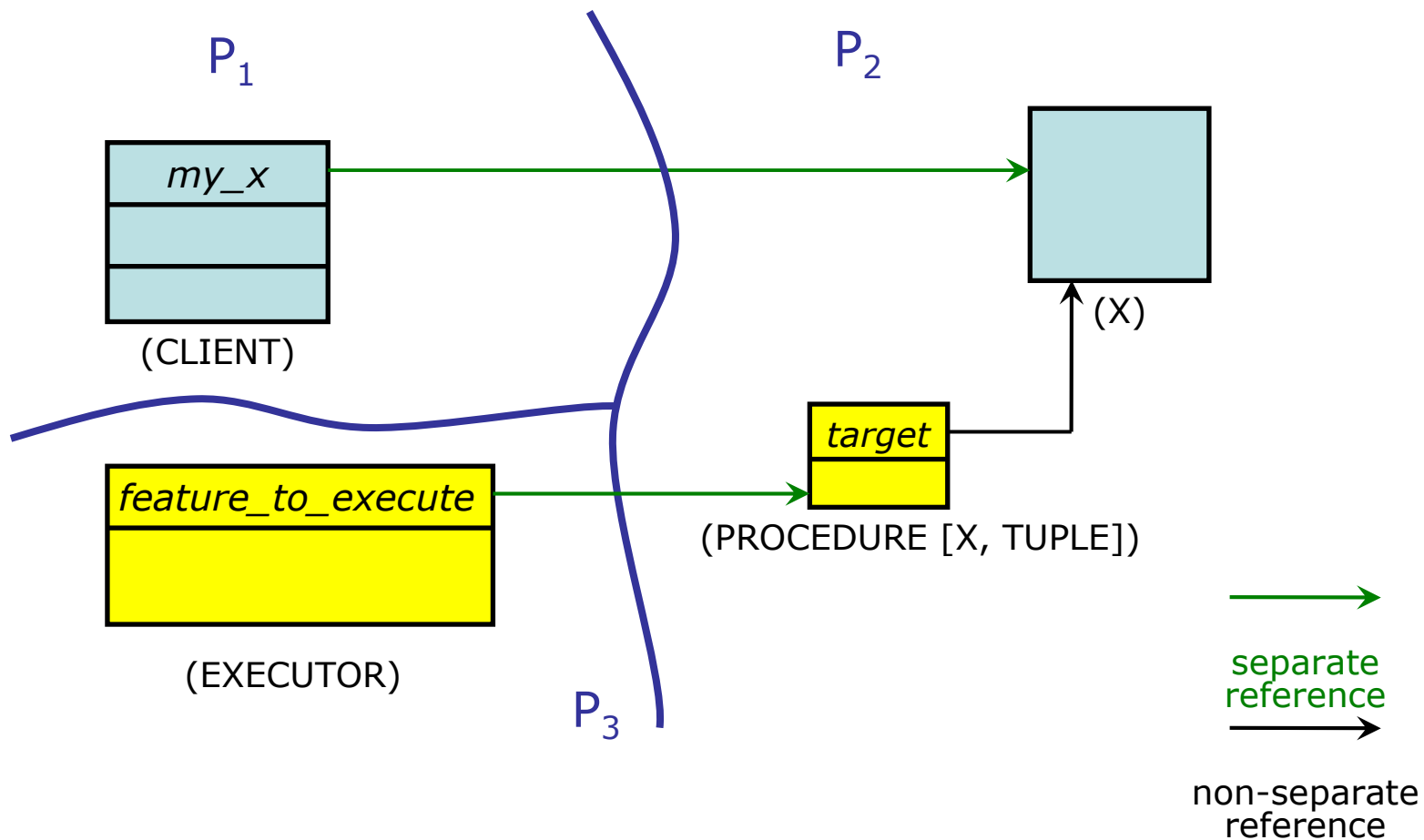
...

end



How to achieve full asynchrony

- *asynch (agent my_x.f)*
- *do_local_stuff*





Separate executors

36

- Feature *asynch* implemented in class *CONCURRENCY*

```
asynch (agent my_x.f)
```

```
asynch (an_agent: ?separate PROCEDURE [ANY, TUPLE])
```

```
-- Call `an_agent' asynchronously.
```

```
-- Note that `an_agent' is not locked.
```

```
local
```

```
  executor: separate EXECUTOR
```

```
do
```

```
  create executor.make (an_agent)
```

```
  launch (executor)
```

```
end
```

- Asynchronous calls on non-separate targets (including **Current**)

```
asynch (agent f)
```

```
-- Call `Current.f' asynchronously.
```

```
-- It will be executed when current processor becomes idle.
```



3rd benefit: waiting faster

37

my_x, my_y: **separate** *X*

...

-- **if** *my_x.b* **or else** *my_y.b* **then** . . . **end**
if *or_else* (*my_x, my_y*) **then** . . . **end**

or_else (*x, y*: **separate** *X*): *BOOLEAN*

do

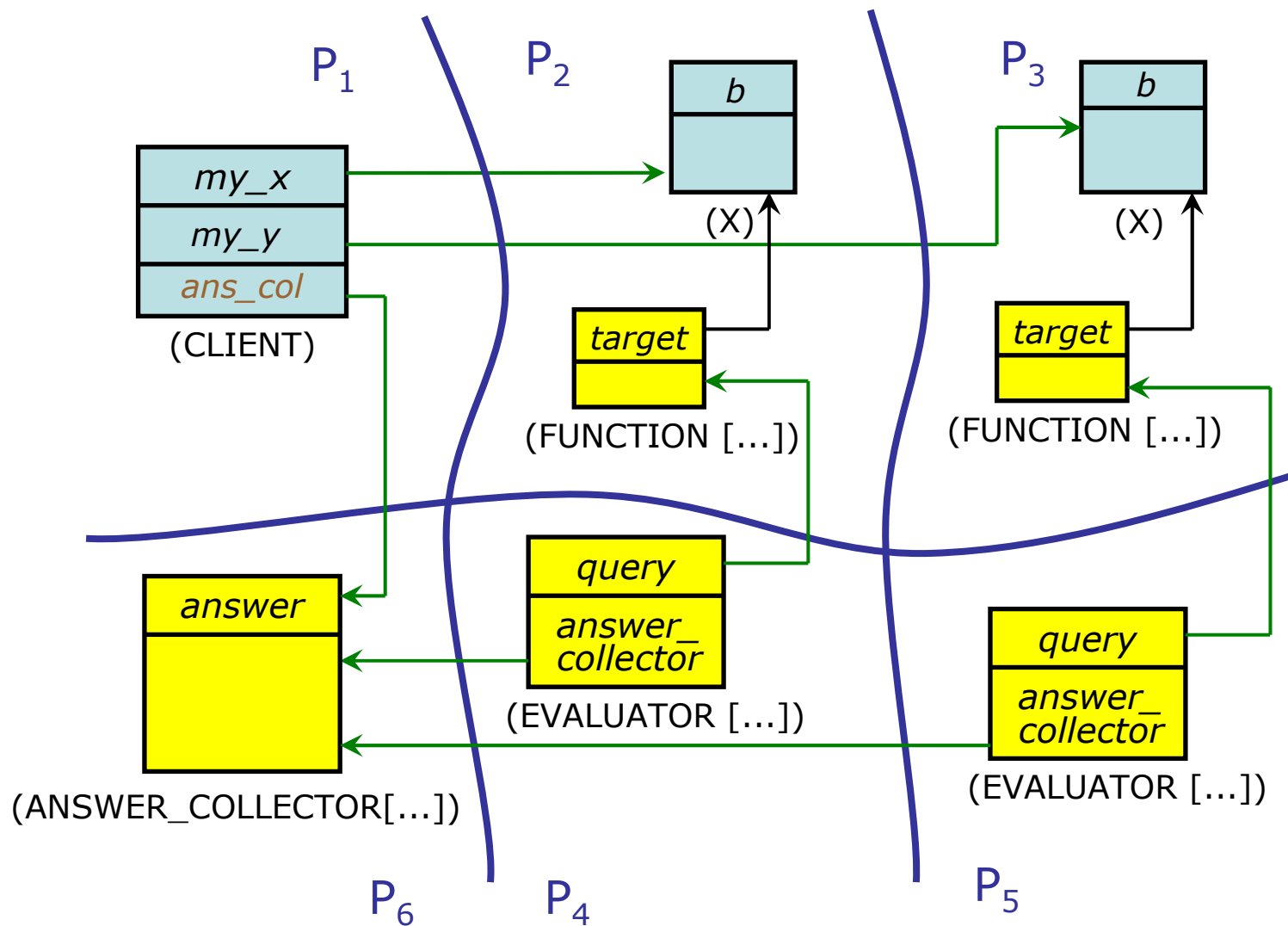
Result := *x.b* **or else** *y.b*

end

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



my_x.b or else my_y.b





Parallel or with agents

39

```
if parallel_or (agent my_x.b, agent my_y.b) then ... End
```

```
parallel_or (a1, a2: ?separate FUNCTION
```

```
          [ANY, TUPLE, BOOLEAN]): BOOLEAN
```

```
-- Result of `a1' or else `a2' computed in parallel.
```

```
local
```

```
  answer_collector: separate ANSWER_COLLECTOR [BOOLEAN]
```

```
do
```

```
  create answer_collector.make (a1, a2)
```

```
  Result := answer (answer_collector)
```

```
end
```

```
answer (a_collector: separate ANSWER_COLLECTOR [ANY]): ANY
```

```
-- Result returned by `an_answer_collector'.
```

```
require
```

```
  answer_ready: a_collector.is_ready
```

```
do
```

```
  Result ?= a_collector.answer
```

```
end
```



Generalised mechanism

40

- Parallel or, parallel and, ...
- Launch n jobs and wait for first result, etc.
- Implemented in class *CONCURRENCY*
- Relies on generic classes
 - *ANSWER_COLLECTOR [RESULT_TYPE]*
 - *EVALUATOR [RESULT_TYPE]*
- For the moment it's a pattern; I'll turn it into a component



Demo

41

make

-- Creation procedure.

local

x, y: **separate** *X*

a, b: **separate** *ORACLE*

do

-- testing asynchrony

create *x*; **create** *y*

asynch (**agent** *x.print_info*)

asynch (**agent** *Current.print_info*)

asynch (**agent** *y.print_info*)

-- testing parallel wait

create *a.make* (**False**, 10000); **create** *b* (**True**, 0)

res := **parallel_or** (**agent** *a.boolean_query*,
agent *b.boolean_query*)

end



Conclusions

42

- **Agents and concurrency**
 - Tricky at first; easy in the end
 - Agents built on separate calls are separate
 - Open-target agents are non-separate on creation
 - 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
 - All these are implemented as library mechanisms
- **Support in scoop2scoopli**
 - **Limited**
 - **You'd need to hack preprocessed code**



That's all, folks!

43

- **SCOOP**
 - Computational model, synchronisation.
 - Traitors, validity rules.
 - Type system for SCOOP, attached and detachable types, lock passing.
 - Advanced O-O techniques: inheritance, polymorphism, genericity, agents.
 - Santa, Santa, Santa, ...

- **Traditional approaches**
 - Safety, liveness, fairness.
 - Atomicity violations, deadlock, livelock.
 - Mutexes, semaphores, barriers, monitors, CCRs, rendezvous.
 - Protected objects, Actors.
 - Active objects, inheritance anomalies.



Questions?

44

- Did I hear "exam"? I knew you'd ask that question. Here's the answer:
- SCOOP (easy questions)
 - Computational model, synchronisation.
 - Traitors, validity rules.
 - Type system for SCOOP, attached and detachable types, lock passing.
 - Advanced O-O techniques: inheritance, polymorphism, genericity, agents.
 - Santa, Santa, Santa, ...
- Traditional approaches (more difficult questions)
 - Safety, liveness, fairness.
 - Atomicity violations, deadlock, livelock.
 - Mutexes, semaphores, barriers, monitors, CCRs, rendezvous.
 - Protected objects, Actors.
 - Active objects, inheritance anomalies.