



---

# Concurrent Object-Oriented Programming

Bertrand Meyer, Piotr Nienaltowski



# Lecture 8 - 9:

## Traitors, validity rules, type system



- Traitors
- Validity rules - first attempt
- Type system for SCOOP
- Handling false traitors



# Refresher: computational model

---

4

- Software system is composed of several **processors**
- Processors are **sequential**; concurrency is achieved through their interplay
- Separate entity denotes a *potentially* **separate object**
- Calls to non-separate objects are **synchronous**
- Calls to separate objects are **asynchronous**



# Refresher: synchronisation

---

5

- Mutual exclusion
  - Locking through **argument passing**
  - Routine **body** is **critical section**
- Condition synchronisation
  - **wait-conditions**
- Re-synchronisation of client and supplier:
  - **wait-by-necessity**
- **Lock passing** through argument passing



## Refresher: separate call rule

---

6

A routine call with separate arguments will execute when all corresponding objects are available

and wait-conditions are satisfied

and hold the objects exclusively for the duration of the routine



# What SCOOP can do for us

7

- Beat *enemy number one* in concurrent world, i.e. atomicity violations
  - Data races
  - Illegal interleavings of calls
- Data races cannot occur in SCOOP.
  - Why? See computational model...
- Separate call rule does not protect us from bad interleaving of calls!
  - How can this happen?



# Traitors here...

```
-- in class C (client)
my_x: separate X
a: A
...
r (x: separate X) is
  do
    a := x.a
  end
...
r (my_x)
```

```
-- supplier
class X
feature
  a: A
end
```

Is this call valid?

**TRAITOR! TRAITOR!**

a.f

And this one?



# Traitors there...

-- in class C (client)

*my\_x*: **separate X**

*a*: A

...

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

**do**

*x.f* (*a*)

**end**

...

*r* (*my\_x*)

-- supplier

**class X**

**feature**

*f* (*a*: A) is

**do**

*a.f*

**end**

**end**

**TRAITOR! TRAITOR!**

And this one?

Is this call valid?



## Consistency rules – first attempt

---

10

- Original model defines four consistency rules that eliminate traitors
  - See OOSC2 chapter 30 for details
- Written in English
- Easy to understand by programmers
- Are they sound? Are they complete?



# SCOOP rules – first attempt

11

## Separateness consistency rule (1)

If the source of an attachment (assignment instruction or argument passing) is separate, its target entity must be separate too.

*r* (*buffer*: **separate** *BUFFER* [*X*]; *x*: *X* ) **is**

**local**

*b1*: **separate** *BUFFER* [*X*]

*b2*: *BUFFER* [*X*]

*x2*: **separate** *X*

**do**

*b1* := *buffer* -- valid

*b2* := *b1* -- invalid

*r* (*b1*, *x2*) -- invalid

**end**



# SCOOP rules – first attempt

12

## Separateness consistency rule (2)

If an actual argument of a separate call is of a reference type, the corresponding formal argument must be declared as separate.

```
store (buffer: separate BUFFER [X]; x: X ) is
```

```
do
```

```
    buffer.put (x)
```

```
end
```

```
-- in class BUFFER [G]
```

```
put (element: separate G) is
```

```
    . . .
```



# SCOOP rules – first attempt

13

## Separateness consistency rule (3)

If the source of an attachment is the result of a separate call to a function returning a reference type, the target must be declared as separate.

```
consume_element (buffer: separate BUFFER [X]) is
```

```
  local
```

```
    element: separate X
```

```
  do
```

```
    element := buffer.item
```

```
    . . .
```

```
  end
```

```
-- in class BUFFER [G]
```

```
item: G is
```

```
  . . .
```



# SCOOP rules – first attempt

14

## Separateness consistency rule (4)

If an actual argument of a separate call is of an expanded type, its base class may not include, directly or indirectly, any non-separate attribute of a reference type.

```
store (buffer: separate BUFFER [X]; x: X ) is
```

```
do
```

```
    buffer.put (x)    -- X must be "fully expanded"
```

```
end
```

```
-- in class BUFFER [G]
```

```
put (element: G) is    -- G is not declared as separate anymore
```

```
...
```



# Problem 1: unsoundness

15

*x*: *X* -- *X* is expanded, see below.

*consume\_element* (*buffer*: **separate** *BUFFER* [*X*]) **is**

**local**

*s*: *STRING*

**do**

*x* := *buffer.item*

traitor!!!

*s* := *x.f.out* -- Valid: call on expanded object.

*s* := *x.g.out* -- Valid! call on separate reference.

**end**

**expanded class** *X*

**feature**

*g*: *STRING*

*f*: *INTEGER* **is** . . .

**end**



## Problem 2: limitations

16

*my\_x*: *X*-- *X* is expanded, see below.

*my\_y*: **separate** *Y*

...

*my\_y.r* (*my\_x*) -- Invalid because *my\_x* is not fully expanded.

**expanded class** *X*

**feature**

*g*: *STRING*

*f*: *INTEGER is . . .*

**end**

-- in class *Y*

*r* (*x*: *X*) **is**

**do**

...

**end**



## Problem 3: more limitations

17

```
class STRING
```

```
feature
```

```
...
```

```
append alias infix "+" (other: like Current) is
```

```
do
```

```
...
```

```
end
```

```
end
```

```
-- in class X
```

```
r (l: separate LIST [STRING]) is
```

```
do
```

```
l.put (l.item (1) + l.item (2))
```

```
end
```

Invalid!!! But  
it should be  
allowed!



## Problem 4: even more limitations

18

*r* (*l*: **separate** *LIST* [*STRING*]) **is**

**local**

*s*: **separate** *STRING*

**do**

*s* := *l.item* (1)

*l.put* (*s*)

**end**

Invalid!!! But it  
should be allowed!



# Let's make it better!

---

19

- SCOOP rules
  - prevent almost all traitors, +
  - are easy to understand by humans, +
  - cannot be directly used by compilers, -
  - not sound, -
  - too restrictive, -
  - no support for *agents*. -
- Can we do better?
  - Refine and **formalise** the rules!



# The question for today

---

20

How do you know whether assignment

$x := y$

--  $x$  and  $y$  are declared as

--  $x: CLASS\_X; y: CLASS\_Y$

and argument passing

$r(x)$

--  $r$  is declared as  $r(an\_x: SOME\_CLASS)$

are valid?

Type system tells you that!



# Type system for SCOOP

---

21

- Prevents all traitors
  - static (compile-time) checks
- Simplifies, refines and formalises SCOOP rules
- Integrates expanded types and agents with SCOOP
  - More about it in lecture 10
- Tool for reasoning about concurrent programs
  - May serve as basis for future extensions, e.g. for deadlock prevention schemes



# Three components of a type

22

- **C** Current processor  
 $x: X$

Some processor (top)  
 $x: \text{separate } X$

- Processor tag  $\alpha \in \{\bullet, \top, \perp, \langle p \rangle, \langle a.\text{handler} \rangle\}$

- Attached/detachable  $\gamma \in \{!, \text{...}\}$

No processor (bottom)  
**Void**

$$\Gamma \vdash x :: (\gamma, \alpha, C)$$



# Examples

23

$x: X$                        $-- x :: (!, \bullet, X)$

$y: \text{separate } Y$                        $-- y :: (!, \top, Y)$

$z: ? \text{ separate } Z$                        $-- z :: (?, \top, Z)$

- Expanded types are attached and non-separate

$i: \text{INTEGER}$                        $-- i :: (!, \bullet, \text{INTEGER})$

- **Void** is detachable                       $-- \text{Void} :: (?, \perp, \text{NONE})$

- **Current** is attached and non-separate

$-- \text{Current} :: (!, \bullet, C_{\text{Current}})$



# Subtyping rules

- Since you don't like Greek letters, I'll keep it informal
- $TT_2 \leq TT_1$  means "TT<sub>2</sub> is a subtype of TT<sub>1</sub>"
- Conformance on class types like in Eiffel, essentially based on inheritance

$$D \leq_{\text{Eiffel}} C \quad \Leftrightarrow \quad (\gamma, \alpha, D) \leq (\gamma, \alpha, C)$$

- Attached  $\leq$  detachable  $(!, \alpha, C) \leq (?, \alpha, C)$
- Any processor tag  $\leq T$   $(\gamma, \alpha, C) \leq (\gamma, T, C)$
- In particular, non-separate  $\leq T$   $(\gamma, \bullet, C) \leq (\gamma, T, C)$
- $\perp \leq$  any processor tag  $(\gamma, \perp, C) \leq (\gamma, \alpha, C)$



## So how does it help us?

25

- We can rely on standard type rules
- Enriched types give us additional guarantees
- Assignment rule: source conforms to target

$$\text{[Assign]} \frac{\Gamma \vdash x :: \Pi_x, \quad \Gamma \vdash e :: \Pi_e, \quad \Gamma \vdash \Pi_e \leq \Pi_x}{\Gamma \vdash x := e}$$

- No need for special validity rules for **separate**



# Is it always that simple?

---

26

- Rules for feature calls are more complex
- “Targettability” of target taken into account
  - Is target attached?
  - Is target’s handler accessible to client’s handler?
- Type of formal arguments depends on type of target



# Unified rules for call validity

27

- Informally, entity  $x$  may be used as target of a feature call in the context of routine  $r$  if and only if  $x$  is **attached** and processor that executes  $r$  has exclusive access to  $x$ 's processor.

## Definition (Targettable entity)

Entity  $x$  is targettable, i.e. it may be used as target of a feature call in the context of routine  $r$ , if and only if  $x$  is of attached type, it is declared with processor tag  $\alpha$  and some attached formal argument of  $r$  has processor tag  $\alpha$ .



## Definition (Valid call)

Call  $x.f(a)$  appearing in the context of routine  $r$  in class  $C$  is valid iff the following conditions hold:

- $x$  is targettable.
- $x$ 's base class has feature  $f$  exported to  $C$ , and actual arguments  $a$  conform in number and type to formal arguments of  $f$ .

$$\frac{\begin{array}{l} \Gamma \vdash e :: TT_e, \quad \Gamma \vdash isTargettable(TT_e), \quad \Gamma \vdash ClassType(TT_e) = C \\ \Gamma \vdash FeatureType(C, f) = TT'_1 \times \dots \times TT'_n \longrightarrow TT_{res} \quad n \geq 0 \\ \Gamma \vdash a_i :: TT_i \quad i \in 1..n, \quad \bar{a} = a_1..a_n, \quad \Gamma \vdash \forall_{i \in 1..n} TT_i \preceq TT_e \otimes TT'_i \end{array}}{\Gamma \vdash e.f(\bar{a}) :: TT_e \star TT_{res}} \quad (\text{T-QCallQual})$$

- **Type combinators** necessary to calculate relative type
  - formal arguments  $\otimes$
  - result  $\star$



# Type combinator \* (result type)

29

$$(\gamma, \alpha, C) * (\delta, \beta, D) = (\delta, \lambda, D)$$

$\beta$			
$\alpha$	$\bullet$	T	$\langle q \rangle$
$\bullet$	$\bullet$	T	T
T	T	T	T
$\langle p \rangle$	$\langle p \rangle$	T	T



# Type combinator $\otimes$ (formals)

30

$$(\gamma, \alpha, C) \otimes (\delta, \beta, D) = (\delta, \lambda, D)$$

$\alpha$	$\beta$		
	$\cdot$	$\top$	$\langle q \rangle$
$\cdot$	$\cdot$	$\top$	$\perp$
$\top$	$\perp$	$\top$	$\perp$
$\langle p \rangle$	$\langle p \rangle$	$\top$	$\perp$



# Expanded types

31

- Both  $*$  and  $\otimes$  preserve expanded types

$$(\gamma, \alpha, C) * (!, \bullet, \text{INTEGER}) = (!, \bullet, \text{INTEGER})$$

$$(\gamma, \alpha, C) \otimes (!, \bullet, \text{BOOLEAN}) = (!, \bullet, \text{BOOLEAN})$$

```
my_x: X      -- my_x :: (!, ●, X)
```

```
my_y: separate Y -- my_y :: (!, T, Y)
```

```
...
```

```
my_y.r (my_x)
```

```
-- (!, ●, X) ≤ (!, T, Y) ⊗ (!, ●, X)
```

```
-- so call is valid
```

```
expanded class X
```

```
feature
```

```
g: STRING
```

```
f: INTEGER is . . .
```

```
end
```



# Expanded types: `deep_import`

---

32

- Expanded objects are non-separate
  - How to prevent traitors?
- Implicit *deep\_import* operation
  - Just like *copy* but it clones (recursively) all non-separate attributes



## Recall: traitors here...

33

```
-- in class C (client)
my_x: separate X
a: A
...
r (x: separate X) is
  do
    a := x.a
  end
```

```
...
r (my_x)
a.f
```

**TRAITOR! TRAITOR!**



# Type system eliminates them

34

-- in class C (client)

*my\_x*: **separate** *X*

-- *my\_x* :: (!, T, X)

*a*: *A* -- *a* :: (!, •, X)

...

*r* (*x*: **separate** *X*) **is** -- *x* :: (!, T, X)

**do**

*a* := *x.a* -- *x.a* :: (!, T, X) \* (!, •, A) = (!, T, A)

-- (!, T, A)  $\not\leq$  (!, •, A)

-- therefore assignment is invalid

**end**

...

*r* (*my\_x*) -- (!, T, X)  $\leq$  (!, •, C)  $\otimes$  (!, T, X) , so call is valid

*a.f*



## Recall: traitors there...

35

```
-- in class C (client)
```

```
my_x: separate X
```

```
a: A
```

```
...
```

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

```
  do
```

```
    x.f (a)
```

```
  end
```

```
...
```

```
r (my_x)
```

**TRAITOR! TRAITOR!**

```
-- supplier
```

```
class X
```

```
feature
```

```
  f (a: A) is
```

```
    do
```

```
      a.f
```

```
    end
```

```
  end
```



# Type system eliminates them

36

```
-- in class C (client)
my_x: separate X
      -- my_x :: (!, τ, X)
a: A   -- a :: (!, •, A)
...
r (x: separate X) is   -- x :: (!, τ, X)
  do
    x.f (a)
    -- (!, •, A) ⊆ (!, τ, X) ⊗ (!, •, A)
    -- (!, •, A) ⊆ (!, ⊥, A)
    -- therefore call is invalid
  end
```

...

```
r (my_x)      (!, τ, X) ⊆ (!, •, C) ⊗ (!, τ, X) , so call is valid
```

```
-- supplier
class X
feature
  f (a: A) is
    -- a :: (!, •, A)
  do
    a.f
  end
end
```



# Explicit processor tags

37

```
p: HANDLER           -- Tag declaration
x: separate <p> X   -- x :: (!, <p>, X)
y: separate <p> Y   -- y :: (!, <p>, Y)
z: separate Z       -- z :: (!, τ, Z)
```

- Attachment

-- Assume that *Y* and *Z* inherit from *X*

```
x := y           -- Valid because (!, <p>, Y) ≤ (!, <p>, X)
```

```
y := z           -- Invalid because (!, τ, Z) ≤ (!, <p>, Y)
```

- Object creation

```
create x         -- Fresh processor is created to handle x.
```

```
create y         -- No new processors created; y is put
                  -- on x's processor.
```



# Implicit processor tags

38

- Declared using “feature” handler on a read-only attached entity (formal argument, **Current**)

```
x: separate <roe.handler> X
-- x is handled by roe's handler.
```

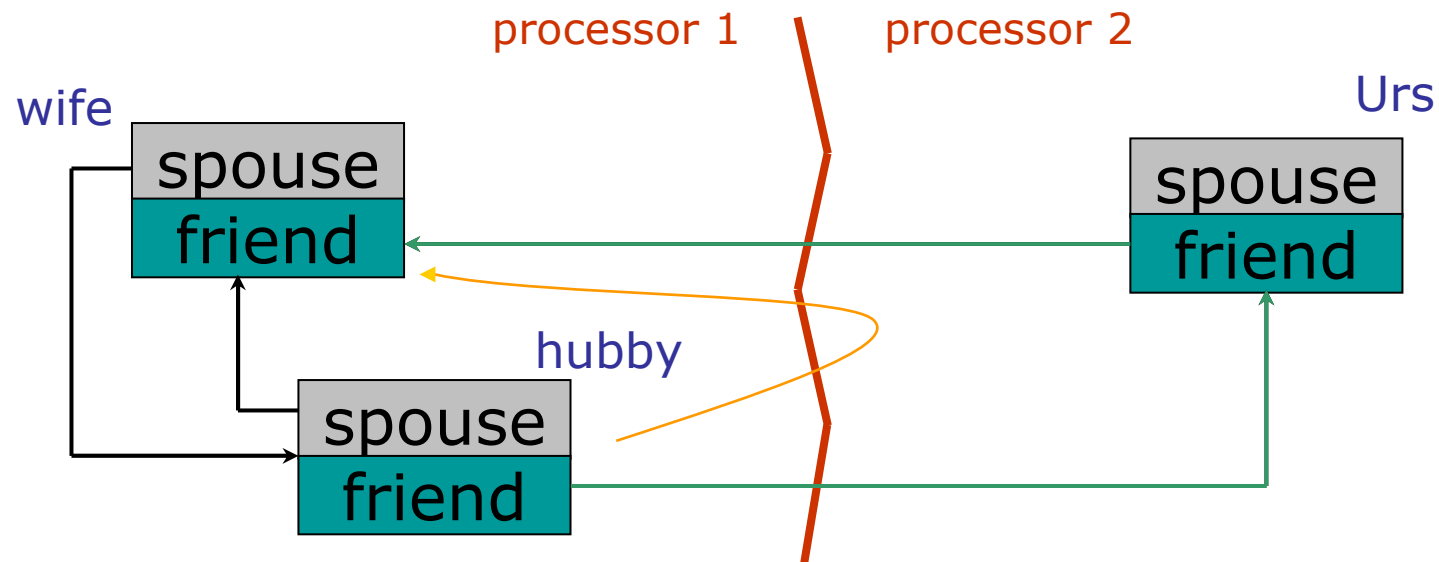
- Attachment, object creation

```
r (list: separate LIST [X])
  local
    s1, s2: separate <list.handler> STRING
    -- s1 :: (!, <list.handler>, STRING), s2 alike
  do
    s1 := list.item (1)
    s2 := list.item (2)
    list.extend (s1 + s2)      -- Valid
    create s1.make_empty    -- s1 created on list's processor
    list.extend (s1)         -- Valid
  end
```



# False traitors

39



*meet\_friend* (*person*: **separate** *PERSON*) **is**

**local**

*a\_friend*: *PERSON*

**do**

*a\_friend* := *person.friend* -- Invalid assignment.

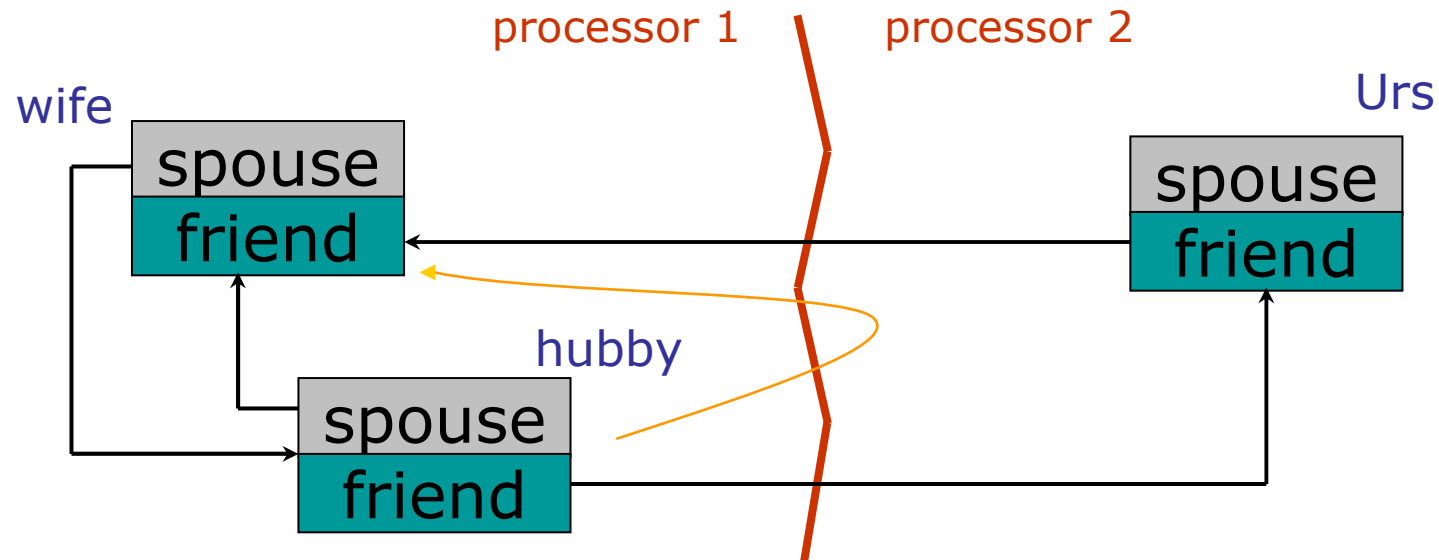
*visit* (*a\_friend*)

**end**



# Handling false traitors

40



*meet\_friend* (*person*: **separate** *PERSON*) **is**

**local**

*a\_friend*: *PERSON*

**do**

*a\_friend* **?=** *person.friend* -- Valid assignment attempt.

**if** *a\_friend* **/= void** **then** *visit* (*a\_friend*) **end**

**end**



# Assignment attempt

---

41

- Like in Eiffel but also “downcasts” processor tags
  - “deep downcast” over expanded attributes
- Eiffel standard has replaced `?=` with **object test**

```
if {a_friend: PERSON} person.friend then  
    visit (a_friend)  
end
```



# That's all, folks!

---

42

Questions?