



SCOOP

Simple Concurrent Object-Oriented Programming



Lecture 3 (9): Type system for SCOOP



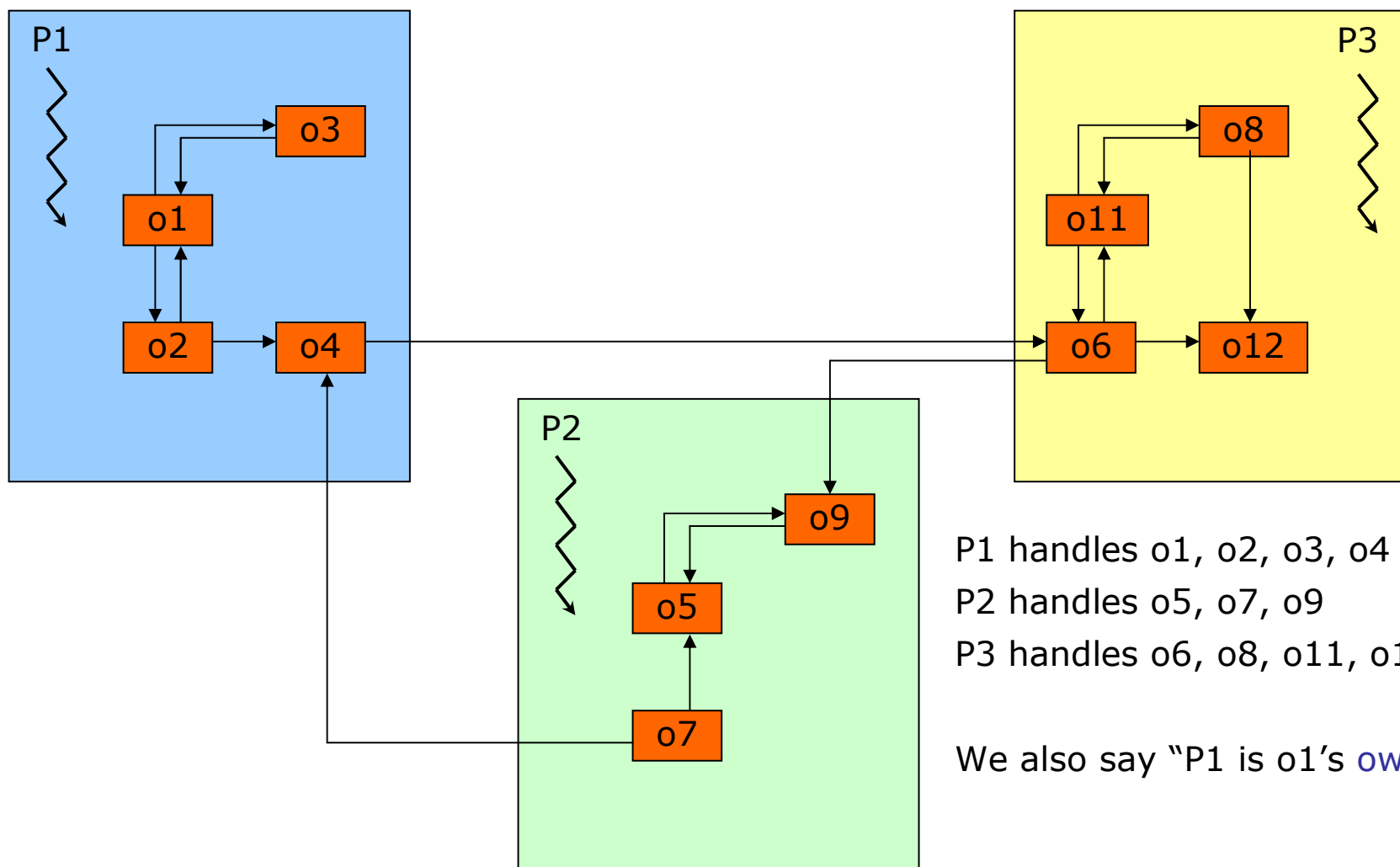
Outline

3

- Refresher on Lecture 2
- Type system for SCOOP
- Validity rules – second attempt
- Examples
- Handling false traitors



Software system



P1 handles o1, o2, o3, o4
P2 handles o5, o7, o9
P3 handles o6, o8, o11, o12

We also say "P1 is o1's owner"



What SCOOP should do for us

5

- Beat *enemy number one* in concurrent world, i.e. data races
 - *Data race occurs when two or more clients concurrently apply some feature on the same supplier.*
- Data races could be caused by so-called **traitors**, i.e. non-separate entities that denote separate objects.

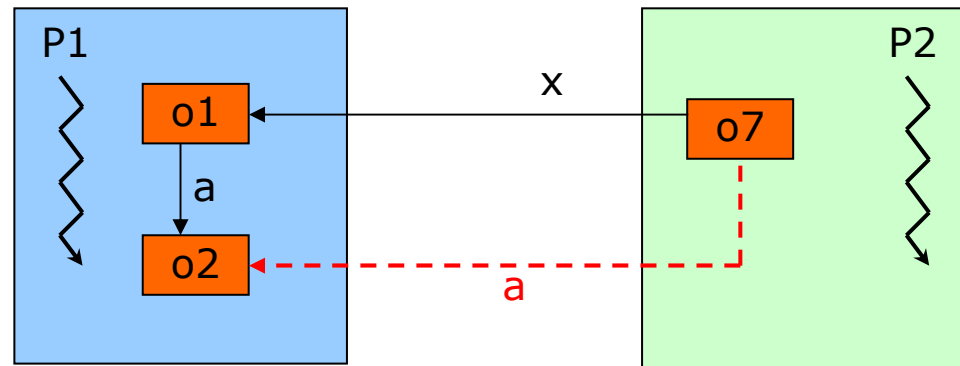


Traitors

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

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

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





Validity rules – first attempt

7

- 4 SCOOP consistency rules
 - prevent data races (almost), +
 - written in English, easy to understand by humans, +
 - cannot be directly used by compilers, -
 - not sound, too restrictive, -
 - no support for *agents*. -
- How to do it better?
 - Refine and **formalise** the rules!



Type system for SCOOP

- Prevents data races
 - static (compile-time) checks
- Simplifies, refines and formalises SCOOP rules
- Integrates expanded types and agents with SCOOP
 - More about it in Lecture 4 (11)
- Ownership-like types
 - Eiffel types augmented with *owner tags*
 - inspired by Peter Mueller's work on applet isolation in JavaCard
- Tool for reasoning about concurrent programs
 - can serve as basis for future extensions (e.g. for deadlock prevention)



A few definitions...

Let *TypeId* denote the set of declared type identifiers of a given Eiffel program. We define the set of tagged types for a given class as

$$\textit{TaggedType} = \textit{OwnerId} \times \textit{TypeId}$$

where *OwnerId* is a set of owner tags declared in the given class. Each class implicitly declares two owner tags: • (*current processor*) and \perp (*unknown*).

The *subtype relation* \prec on tagged types is the smallest reflexive, transitive relation satisfying the following axioms, where α is a tag, $S, T \in \textit{TypeId}$, and $\prec_{\textit{Eiffel}}$ denotes the subtype relation on *TypeId*:

$$(\alpha, T) \prec (\alpha, S) \Leftrightarrow T \prec_{\textit{Eiffel}} S$$

$$(\alpha, T) \prec (\perp, T)$$



Specifying the type

10

class *C*

owner

r1, r2 -- owner tags. \bullet and \perp are declared implicitly.

feature

a: *A* -- *a* :: (\bullet , *A*)

x: **separate** *X* -- *x* :: (\perp , *X*)

y: **separate** *Y* **within** *r1* -- *y* :: (*r1*, *Y*)

r (*an_x*: **separate** *X*) **is** -- *an_x* :: (\perp , *X*)

do

a := *an_x.a*

end

...

end



And off we go!

11

[Current]

$$\frac{}{\Gamma \vdash \mathbf{Current} :: (\bullet, T_{Current})}$$

[DecNS]

$$\frac{l : T \in \Gamma}{\Gamma \vdash l :: (\bullet, T)}$$

[DecSU]

$$\frac{l : \mathbf{separate} \ T \in \Gamma}{\Gamma \vdash l :: (\perp, T)}$$

[DecSO]

$$\frac{l : \mathbf{separate} \ T \ \mathbf{within} \ r \in \Gamma}{\Gamma \vdash l :: (r, T)}$$



Assignment

12

$$\text{[Assign]} \frac{\Gamma \vdash l :: (\alpha, T), \Gamma \vdash e :: (\beta, S), (\beta, S) \prec (\alpha, T)}{\Gamma \vdash l := e}$$



Separateness consistency rule (1)

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



Feature call

13

$$\text{[QCall]} \frac{\Gamma \vdash x :: (\alpha, T), \quad \Gamma \vdash a :: (\beta, S), \quad (\beta, S) \prec (\alpha_{fa}, T_{fa})}{\Gamma \vdash x.f(a) :: (\alpha_{fr}, T_{fr})}$$

FormalArg is the set of formal arguments of the routine where the expression is evaluated, (α_{fa}, T_{fa}) is the type of the formal argument of feature f (we assume here that f has only one argument), (α_{fr}, T_{fr}) is the type of its result.

We also define the **type combinator**

$$* : \text{TaggedType} \times \text{TaggedType} \rightarrow \text{TaggedType}$$

$$(\alpha, T) * (\beta, S) = \begin{cases} (\beta, S) & \text{if } \alpha = \bullet \\ (\alpha, S) & \text{if } \beta = \bullet \\ (\perp, S) & \text{otherwise} \end{cases}$$



Type combinator

$$(\alpha, T) * (\beta, S) = (\gamma, S)$$

β			
α	\bullet	\perp	$r2$
\bullet	\bullet	\perp	$r2$
\perp	\perp	\perp	\perp
$r1$	$r1$	\perp	\perp

non-separate
calls preserve
owner tag

$x :: (\alpha, T)$

$f :: (\beta, S)$

$x.f :: (\gamma, S)$

separate calls
always return
separate type



And now less formally

Separate Call rule

$$\begin{array}{c}
 \Gamma \vdash x :: (\alpha, T), \quad \Gamma \vdash a :: (\beta, S), \quad (\beta, S) \prec (\alpha, T) * (\alpha_{fa}, T_{fa}) \\
 \text{[QCall]} \quad \frac{\alpha \neq \bullet \Rightarrow x \in \text{FormalArg}, \quad \alpha = \perp \Rightarrow \alpha_{fa} = \perp}{\Gamma \vdash x.f(a) :: (\alpha, T) * (\alpha_{fr}, T_{fr})}
 \end{array}$$

consistency rules 1 and 2

$x.f(a)$

together with [Assign] implies consistency rule 3

FormalArg is the set of formal arguments of the routine where the expression is evaluated, (α_{fa}, T_{fa}) is the type of the formal argument of feature f (we assume here that f has only one argument), (α_{fr}, T_{fr}) is the type of its result.

$$(\alpha, T) * (\beta, S) = \begin{cases} (\beta, S) & \text{if } \alpha = \bullet \\ (\alpha, S) & \text{if } \beta = \bullet \\ (\perp, S) & \text{otherwise} \end{cases}$$



Attachment rule (0)

The type of the source of an attachment (assignment instruction or argument passing) must conform to the type of its target.

$$x :: (\alpha, T), \quad y :: (\beta, S)$$
$$x := y \text{ valid iff } (\beta, S) \prec (\alpha, T)$$

- But this attachment rule that already exists in the type system!
- The programmer just applies standard rule with augmented types.



Expression type rule (1)

Type of an expression (query call) $x.f$ depends on the type of its target ($x :: (\alpha, T)$) and the declared type of the query ($f :: (\beta, S)$).

$$x.f :: (\alpha, T) * (\beta, S)$$

$$(\alpha, T) * (\beta, S) = \begin{cases} (\beta, S) & \text{if } \alpha = \bullet \\ (\alpha, S) & \text{if } \beta = \bullet \\ (\perp, S) & \text{otherwise} \end{cases}$$



Consistency rules – second attempt

18

Fully expanded types rule (2)

An entity x that represents an object of a fully expanded type FT (i.e. whose base class does not include, directly or indirectly, any non-separate attribute of reference) is seen as non-separate in any typing context.

$$x :: (\bullet, FT)$$

Basic types *INTEGER*, *BOOLEAN*, *CHARACTER*, *REAL*, etc. are fully expanded.



Examples: attachment rule

19

class X

owner

$r1, r2$

feature

$a: X$ -- $a :: (\bullet, X)$

$x: \text{separate } X$ -- $x :: (\perp, X)$

$y: \text{separate } X \text{ within } r1$ -- $y :: (r1, X)$

$z: \text{separate } Z \text{ within } r1$ -- $z :: (r1, Z)$

$x := a$

-- valid because (\bullet, X) is a subtype of (\perp, X)

$a := x$

-- invalid because (\perp, X) is not a subtype of (\bullet, X)

end



Examples: attachment rule

20

class X

owner

$r1, r2$

feature

$a: X$ -- $a :: (\bullet, X)$

$x: \text{separate } X$ -- $x :: (\perp, X)$

$y: \text{separate } X \text{ within } r1$ -- $y :: (r1, X)$

$z: \text{separate } Z \text{ within } r1$ -- $z :: (r1, Z)$

$x := y$

-- valid because $(r1, X)$ is a subtype of (\perp, X)

$y := x$

-- invalid because (\perp, X) is not a subtype of $(r1, X)$

end



Examples: attachment rule

21

class X

owner

$r1, r2$

feature

$a: X$ **--** $a :: (\bullet, X)$

$x: \text{separate } X$ **--** $x :: (\perp, X)$

$y: \text{separate } X \text{ within } r1$ **--** $y :: (r1, X)$

$z: \text{separate } Z \text{ within } r1$ **--** $z :: (r1, Z)$

$y := a$

-- invalid because (\bullet, X) is not a subtype of $(r1, X)$

$a := y$

-- invalid because $(r1, X)$ is not a subtype of (\bullet, X)

end



Examples: attachment rule

22

class X

owner

$r1, r2$

feature

$a: X$ *-- $a :: (\bullet, X)$*

$x: \text{separate } X$ *-- $x :: (\perp, X)$*

$y: \text{separate } X \text{ within } r1$ *-- $y :: (r1, X)$*

$z: \text{separate } Z \text{ within } r1$ *-- $z :: (r1, Z)$*

-- assume that Z is a descendant of X

$y := z$

-- valid because $(r1, Z)$ is a subtype of $(r1, X)$

$z := y$

-- invalid because $(r1, X)$ is not a subtype of $(r1, Z)$

end



Examples: attachment rule

23

```
class X
owner
  r1, r2
feature
a: A    -- a :: (•, A)
x: separate X      -- x :: (⊥, X)
y: separate X within r1  -- y :: (r1, X)
z: separate Z within r1  -- z :: (r1, Z)

r (an_x: separate X) is      -- an_x :: (⊥, X)
  do ... end

  r (a)
-- valid because (•, X) is a subtype of (⊥, X)

end
```



Examples: attachment rule

24

class X

owner

$r1, r2$

feature

$a: A$ **--** $a :: (\bullet, A)$

$x: \text{separate } X$ **--** $x :: (\perp, X)$

$y: \text{separate } X \text{ within } r1$ **--** $y :: (r1, X)$

$z: \text{separate } Z \text{ within } r1$ **--** $z :: (r1, Z)$

r ($an_x: \text{separate } X$) **is** **--** $an_x :: (\perp, X)$

do ... end

$r(x)$

-- valid because (\perp, X) is a subtype of (\perp, X)

end



Examples: attachment rule

25

```
class X
owner
  r1, r2
feature
a: A    -- a :: (•, A)
x: separate X      -- x :: ( $\perp$ , X)
y: separate X within r1  -- y :: (r1, X)
z: separate Z within r1  -- z :: (r1, Z)

r (an_x: separate X) is      -- an_x :: ( $\perp$ , X)
  do ... end

  r (z)
-- valid because (r1, Z) is a subtype of ( $\perp$ , X)

end
```



Examples: attachment rule

26

class X

owner

$r1, r2$

feature

$a: A$ **--** $a :: (\bullet, A)$

$x: \text{separate } X$ **--** $x :: (\perp, X)$

$y: \text{separate } X \text{ within } r1$ **--** $y :: (r1, X)$

$z: \text{separate } Z \text{ within } r1$ **--** $z :: (r1, X)$

$s (an_x: X) \text{ is}$ **--** $an_x :: (\bullet, X)$

do ... end

$s (x)$

-- invalid because (\perp, X) is not a subtype of (\bullet, X)

end



Examples: attachment rule

27

class X

owner

$r1, r2$

feature

$a: A$ **--** $a :: (\bullet, A)$

$x: \text{separate } X$ **--** $x :: (\perp, X)$

$y: \text{separate } X \text{ within } r1$ **--** $y :: (r1, X)$

$z: \text{separate } Z \text{ within } r1$ **--** $z :: (r1, Z)$

$s (an_x: X) \text{ is}$ **--** $an_x :: (\bullet, X)$

do ... end

$s (a)$

-- valid because (\bullet, X) is a subtype of (\bullet, X)

end



Examples: expression type rule

28

class X

owner

$r1, r2$

feature

$a: A$ **--** $a :: (\bullet, A)$

$x: \text{separate } X$ **--** $x :: (\perp, X)$

$y: \text{separate } X \text{ within } r1$ **--** $y :: (r1, X)$

$r (an_x: \text{separate } X) \text{ is}$ **--** $an_x :: (\perp, X)$

do ... end

$a := x.a$

-- invalid because (\perp, A) is not a subtype of (\bullet, A)

$x := y.x$

-- valid because (\perp, X) is a subtype of (\perp, X)

end



Examples: fully expanded types rule

29

class *X*

owner

r1, r2

feature

i: *INTEGER* -- *i* :: (\bullet , *INTEGER*)

x: **separate** *X* -- *x* :: (\perp , *X*)

s (*an_i*: *INTEGER*) **is** -- *an_i* :: (\bullet , *INTEGER*)

do ... end

s (*i*) -- obviously valid

s (*x.i*)

-- valid because *x.i* :: (\bullet , *INTEGER*)

x.s (*i*)

-- valid

end

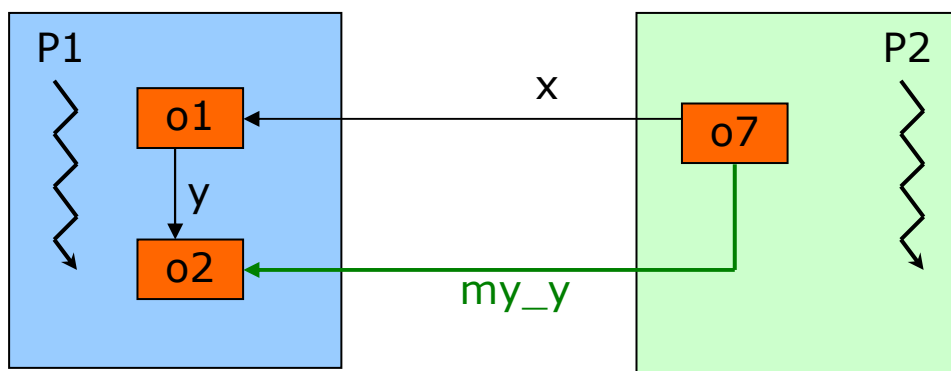


Why do we need explicit owners?

30

```
class X
feature
  y: Y
  set_y (a_y: Y) is
    do
      y := a_y
    end
end
```

```
-- in class C
r (x: separate X) is
  local
    my_y: separate Y
  do
    my_y := x.y
    x.set_y (my_y) -- oops...
    -- set_y takes non-separate
    -- formal argument!
  end
```





Specifying owners

31

- We need the possibility to say:

Entities x_1, x_2, \dots, x_n denote objects that are handled by the same processor.

We say that x_1, x_2, \dots, x_n *are owned by the same processor.*

```
class X
feature
  y: Y
  set_y (a_y: Y) is
    do
      y := a_y
    end
end
```

```
-- in class C
owner r1
feature
  r (x: separate X  $\in$  r1) is
    local
      my_y: separate Y  $\in$  r1
    do
      my_y := x.y
      x.set_y (my_y) -- it's alright now!
    end
```



Another use of owner tags

32

class X

owner

$r1, r2$

feature

x : **separate** X $-- x :: (\perp, X)$

y, z : **separate** X **within** $r1$ $-- y :: (r1, X), -- z :: (r1, X)$

create x $--$ on which processor is x created?

$--$ some fresh processor

create y

$-- y$ is created on (a fresh) processor denoted by $r1$

create z

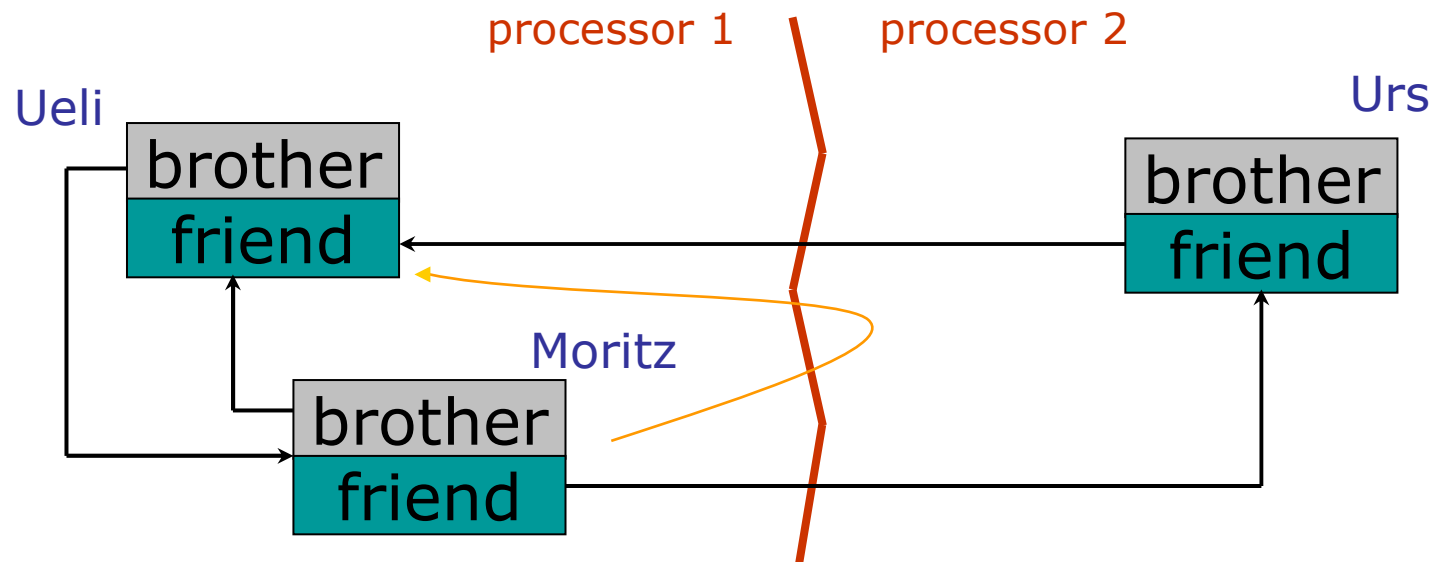
$-- z$ is created on processor denoted by $r1$

end



Handling false traitors

34



meet_someone_elses_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



Semantics of assignment attempt

35

instruction **$l \ ?= \ e$**

with **static types** $l :: (\alpha, T)$, $e :: (\beta, S)$

and **dynamic type** $e :: (\beta_d, S_d)$

is “equal” to:

if $(\beta_d, S_d) \prec (\alpha, T)$ **then** $l \leftarrow e$ **else** $l \leftarrow \mathbf{void}$ **end**

- Like in Eiffel but also downcasts *owner tag*
 - “deep downcast” over expanded attributes



What we have learnt today

36

Let *TypeId* denote the set of declared type identifiers of a given Eiffel program. We define the set of tagged types for a given class as

$$\textit{TaggedType} = \textit{OwnerId} \times \textit{TypeId}$$

where *OwnerId* is a set of owner tags declared in the given class. Each class implicitly declares two owner tags: • (*current processor*) and \perp (*undefined*).

The *subtype relation* \prec on tagged types is the smallest reflexive, transitive relation satisfying the following axioms, where α is a tag, $S, T \in \textit{TypeId}$, and $\prec_{\textit{Eiffel}}$ denotes the subtype relation on *TypeId*:

$$(\alpha, T) \prec (\alpha, S) \Leftrightarrow T \prec_{\textit{Eiffel}} S$$

$$(\alpha, T) \prec (\perp, T)$$



What we have learnt today

37

class C

owner

$r1, r2$ -- owner tags. \bullet and \perp are declared implicitly.

feature

$a: A$ -- $a :: (\bullet, A)$

$x: \text{separate } X$ -- $x :: (\perp, X)$

$y: \text{separate } Y \text{ within } r1$ -- $y :: (r1, Y)$

$r (an_x: \text{separate } X) \text{ is}$ -- $an_x :: (\perp, X)$

do

$a := an_x.a$

end

...

end



What we have learnt today

38

Attachment rule (0)

The type of the source of an attachment (assignment instruction or argument passing) must conform to the type of its target.

$$x :: (\alpha, T), \quad y :: (\beta, S)$$
$$x := y \text{ valid iff } (\beta, S) \prec (\alpha, T)$$



What we have learnt today

39

Expression type rule (1)

Type of an expression (query call) $x.f$ depends on the type of its target ($x :: (\alpha, T)$) and the declared type of the query ($f :: (\beta, S)$).

$$x.f :: (\alpha, T) * (\beta, S)$$

$$(\alpha, T) * (\beta, S) = \begin{cases} (\beta, S) & \text{if } \alpha = \bullet \\ (\alpha, S) & \text{if } \beta = \bullet \\ (\perp, S) & \text{otherwise} \end{cases}$$



What we have learnt today

40

Fully expanded types rule (2)

An entity x that represents an object of a fully expanded type FT (i.e. whose class does not declare any non-separate attribute of reference or not-fully-expanded type) is seen as non-separate in any typing context Γ .

Basic types *INTEGER*, *BOOLEAN*, *CHARACTER*, *REAL*, etc. are fully expanded.



That's all, folks!

41

Questions?