ETH Zürich

Chair of Software Engineering

Concepts of Concurrent Computation

Bertrand Meyer
Sebastian Nanz

Lecture 7: SCOOP type system

Traitor

A traitor is an entity that

- Statically, is declared as non-separate
- During an execution, can become attached to a separate object

Traitors here…

```
-- In class C (client)
x1: separate T
a: A

r (x: separate T)
      do
         a := x.b
      end

r (x1)
a.f
```

```
-- Supplier
class T feature
         b: A
end
```

Is this call valid? ✓

And this one? ✗

Traitors there...

```
-- In class C (client)
x1: separate T
a: A

r (x: separate T)
      do
            x.f (a)
      end

r (x1)
```

Is this call valid? ✓

```
-- Supplier
class T feature
      f (b: A)
            do
                  b.f
            end
end
```

And this one? ✗

Consistency rules: first attempt

Original model (Object-Oriented Software Construction, Chapter 30) defines four consistency rules that eliminate traitors

Written in English

Easy to understand by programmers

Are they sound? Are they complete?

Consistency rules: first attempt

Separateness Consistency Rule (1)

If the source of an attachment (assignment or argument passing) is separate, its target must also be separate

```
r (buf: separate BUFFER [T]; x: T )
    local
        buf1: separate BUFFER [T]
        buf2: BUFFER [T]
        x2: separate T
    do
        buf1 := buf          -- Valid
        buf2 := buf1         -- Invalid
        r (buf1, x2)         -- Invalid
    end
```

Consistency rules: first attempt

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

```
        -- In class BUFFER [G]:
put (element: separate G)

        -- In another class:
store (buf: separate BUFFER [T]; x: T)
    do
        buf.put (x)
    end

    . . .
```

Consistency rules: first attempt

Separateness Consistency Rule (3)

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

```
    -- In class BUFFER [G]:
item: G

        -- In another class:
consume (buf: separate BUFFER [T])
    local
        element: separate T
    do
        element := buf.item
        . . .
    end
```

(*A query is an attribute or function)

Consistency rules: first attempt

Separateness Consistency Rule (4)

If an actual argument or result 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.

```
    -- In class BUFFER [G]:
put (element: G)
    -- G not  declared separate

    -- In another class:
store (buf: separate BUFFER [E]; x: E)
    do
        buf.put (x)
            -- E must be "fully expanded"
    end

    . . .
```

The “ad hoc” rules are too restrictive

```
r (l: separate LIST [STRING])
    local
        s: separate STRING
    do
        s := l [1]
        l.put (s)            -- Invalid according to Rule 2
                                -- but is harmless
    end
```

Ad hoc SCOOP rules: assessment

The rules

- Prevent almost all traitors, +
- Are easy to understand by humans, +
- No soundness proof, -
- Too restrictive, -
- No support for agents -

Can we do better?

- Refine and formalize the rules

A type system for SCOOP

Goal: prevent all traitors through static (compile-time) checks

Simplifies, refines and formalizes ad hoc rules

Integrates expanded types and agents

Three components of a type

Notation:

$\Gamma \;|\!-\; x : (\gamma, \alpha, C)$

Under the binding Γ,
x has the type (γ, α, C)

1. Attached/detachable: $\gamma \in \{!, ?\}$

Some processor (top)
x: **separate** U

Current processor

2. Processor tag $\alpha \in$ {•, ⊤, ⊥, <p>, <a.handler>}

3. Ordinary (class) type *C*

No processor (bottom)

Examples

u: U -- u : (!, •, U)

v: **separate** V -- v : (!, T, V)

w: **detachable separate** W -- w : (?, T, W)

-- Expanded types are attached and non-separate:

i: INTEGER -- i : (!, •, INTEGER)

Void -- Void : (?, ⊥, NONE)

Current -- Current : (!, • , Current)

x: **separate** <px> T -- x : (!, px, T)

y: **separate** <px> Y -- y : (!, px, Y)

z: **separate** <px> Z -- z : (!, px, Z)

Subtyping rules

Conformance on class types like in Eiffel, essentially based on inheritance:

$$D \preceq_{Eiffel} C \quad \Leftrightarrow \quad (\gamma, \alpha, D) \preceq (\gamma, \alpha, C)$$

Standard Eiffel (non-SCOOP) conformance

Attached $\preceq$ detachable:

$$(!, \alpha, C) \preceq (?, \alpha, C)$$

Any processor tag $\preceq \top$:

$$(\gamma, \alpha, C) \preceq (\gamma, \top, C)$$

In particular, non-separate $\preceq \top$:

$$(\gamma, \bullet, C) \preceq (\gamma, \top, C)$$

$\bot \preceq$ any processor tag:

$$(\gamma, \bot, C) \preceq (\gamma, \alpha, C)$$

Using the type rules

We can rely on the standard approach to assess validity

- Assignment rule: source conforms to target

Enriched types give us additional guarantees

No need for special validity rules for separate variables and expressions

Assignment examples

```
a: separate C                              -- a : (!, T, C)
b: C                                       -- b : (!, •, C)
c: detachable C                            -- c : (?, •, C)
f (x, y: separate C) do ... end            -- x : (!, T, C), y : (!, T, C)
g (x: C) do ... end                        -- x : (!, •, C)
h (x: detachable C): separate <p> C        -- x : (?, •, C) : (!, p, C)
    do ... end
```

f (a, b) ✓

f (a, c) — Invalid

g (a) — Invalid

a := h (b) ✓

a := h (a) — Invalid

Unified rules for call validity

Informally, a variable x may be used as target of a separate feature call in a routine r if and only if:

- x is attached
- The processor that executes r has exclusive access to x's processor

Feature call rule

An expression exp of type (d, p, C) is **controlled** if and only if exp is attached and satisfies *any* of the following conditions:

- exp is non-separate, i.e. p = •
- exp appears in a routine r that has an attached formal argument a with the same handler as exp, i.e. p = a.handler

A call x.f (a) appearing in the context of a routine r in a class C is valid if and only if *both*:

- x is controlled
- x's base class exports feature f to C, and the actual arguments conform in number and type to formal arguments of f

Unqualified explicit processor tags

Unqualified explicit processor tags rely on a processor attribute.

- p: PROCESSOR -- Tag declaration
- x: **separate** <p> T-- x : (!, <p>, T)
- y: **separate** <p> Y -- y : (!, <p>, Y)
- z: **separate** Z -- z : (!, T, Z)

Attachment (where Y is a descendant of T, and Z a descendant of Y)

- x := y -- Valid because (!, <p>, Y) ≤ (!, <p>, T)
- y := z -- Invalid because (!, T, Z) ≰ (!, <p>, Y)

Object creation

- **create** x -- Fresh processor created to handle x.
- **create** y -- No new processors created; y is put
  -- on x's processor.

Object creation

```
p: PROCESSOR

a: separate X
b: X
c, d: separate <p> X

create a

create b

create c

create d
```

Processor tag

Create fresh processor for a

Place b on current processor

Create fresh processor p for c

Processor p already exists: place d on p

Qualified explicit processor tags

Declared using "feature" handler on a read-only attached entity (such as a formal argument or current object)

```
x: separate <y.handler> T
        -- x is handled by handler of  y
```

Attachment, object creation:

```
r (list: separate LIST [T])
  local
    s1, s2: separate <list.handler> STRING
                        -- s1, s2 : (!, <list.handler>, STRING)
  do
        s1 := list [1]
        s2 := list [2]
        list.extend (s1 + s2)          -- Valid
        create s1.make_empty           -- s1 created on list's processor
        list.extend (s1)               -- Valid
end
```

Processor tags

Processor tags are always **relative** to the current object

For example, an entity declared as non-separate is seen as non-separate by the current object. Separate clients, however, should see the entity as separate, because from their point of view it is handled by a different processor

Type combinators are necessary to calculate the (relative) type of:

- Formal arguments
- Result

Result type combinator

What is the type T_{result} of a query call x.f (...)?

$T_{result} = T_{target} * T_f$

$= (\alpha x, px, TX) * (\alpha f, pf, TF)$

$= (\alpha f, \mathbf{pr}, TF)$

x * f

px \ pf	•	T	<q>
•	•	T	T
T	T	T	T
<p>	<p>	T	T

Argument type combinator

What is the expected actual argument type in x.f (a)?

$$T_{actual} = T_{target} \otimes T_{formal}$$
$$= (\alpha x, px, TX) \otimes (\alpha f, pf, TF)$$
$$= (\alpha f, \mathbf{pa}, TF)$$

x ⊗ f

px \ pf	•	T	‹q›
•	•	T	⊥
T	⊥	T	⊥
‹p›	‹p›	T	⊥

Type combinators and expanded types

Expanded objects are always attached and non-separate.

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

- $(\gamma, \alpha, C) * (!, \bullet, INTEGER) = (!, \bullet, INTEGER)$
- $(\gamma, \alpha, C) \otimes (!, \bullet, BOOLEAN) = (!, \bullet, BOOLEAN)$

```
x1: EXP                 -- x1 : (!, •, EXP)
y1: separate Y          -- y1 : (!, T, Y)
y1.r (x1)               -- (!, •, EXP) ≤ (!, T, Y) ⊗ (!, •, EXP)
                        -- so call is valid
```

```
expanded class
    EXP
feature
    g: STRING
    f: INTEGER
end
```

```
r (a: EXP) do … end
```

Type combinators and expanded types

The non-separateness of expanded objects needs to be preserved when such an object crosses processor barriers.

Import operation (implicit): like copy, but clones (recursively) all non-separate attributes.

Variations

- Deep import: The relative separateness of objects is preserved; copies are placed on the same processors as their originals.
- Flat import: The whole object structure is placed on the client's processor.
- Independent import: The relative separateness of objects is preserved but copies are placed on fresh processors.

Recall: Traitors here...

```
-- in class C (client)
x1: separate T          -- x1 : (!, T, T)
a: A                    -- a : (!, •, A)

r (x: separate T)       -- x : (!, T, T)
        do
            a := x.a
        end

r (x1)
a.f                     -- Traitor
```

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

(!, T, A) ≰ (!, •, A)

So assignment is invalid

```
-- Supplier
class T                 -- a : (!, •, A)
feature
        a: A
end
```

Recall: Traitors there…

```
-- in class C (client)
x1: separate Z
b: A

r (x: separate Z)
      do
         x.f (b)
      end

r (x1)
```

x1 : (!, T, Z)

b : (!, •, A)

x : (!, T, Z)

(!, •, A) ≤ (!, T, Z) ⊗ (!, •, A)

(!, •, A) ≰ (!, ⊥, A)

So call is invalid

```
-- supplier
class Z
feature
      f (a: A)
         do
            a.f
         end
end
```

a : (!, •, A)

Traitor

Implicit types

An attached non-writable entity e of type T_e = (!, α, C) also has an implicit type $T_{e\ imp}$ = (!, e.handler, C).

Example

x :: (!, T, T) = (!, x.handler, T)

r (x: **separate** T; y: **detachable** Y)

y :: (?, •, Y) no implicit type because y is detachable

local

z: **separate** Z

z :: (!, T, Z) no implicit type because z is writable

do ... **end**

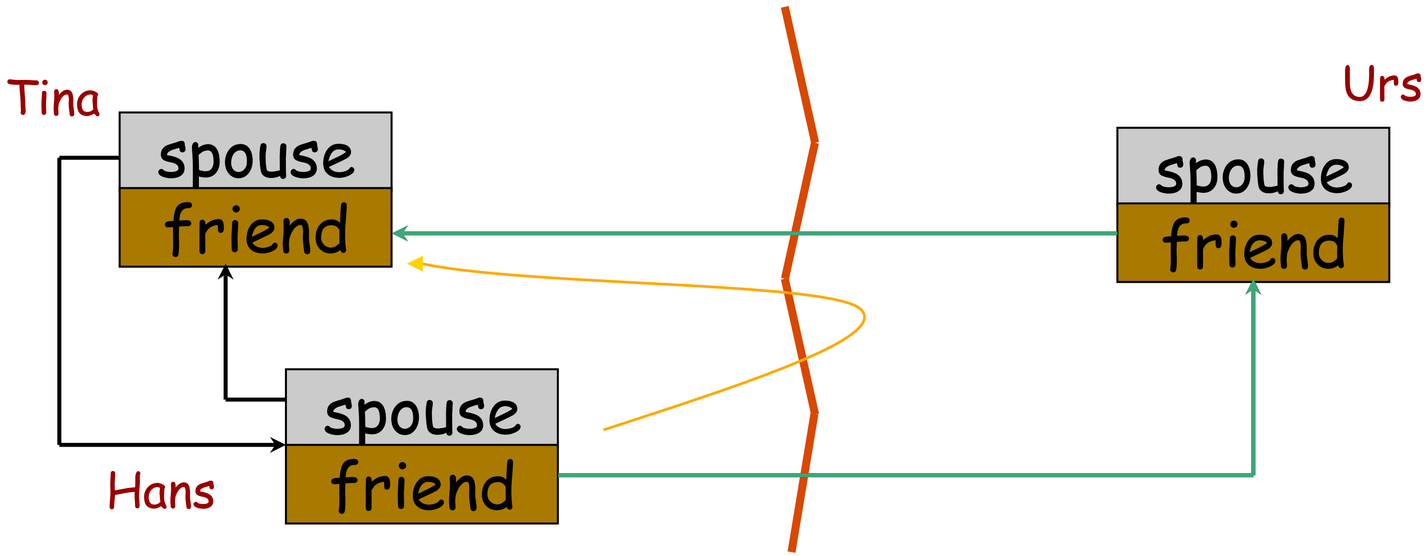
s :: (!, •, STRING) = (!, s.handler, STRING)

s: STRING = "I am a constant"

u :: (!, T, U) = (!, u.handler, U)

u: **separate** U **once** ... **end**

False traitors




```
meet_friend (p: separate PERSON)
    local
        a_friend: PERSON
    do
        a_friend := p.friend     -- Invalid
        visit (a_friend)
    end
```

```
visit (p: PERSON)
    do ... end
```

Hans.meet_friend (Urs)

Handling false traitors with object tests

Use Eiffel object tests with downcasts of processor tags.

An object test succeeds if the run-time type of its source conforms in all of

- Detachability
- Locality
- Class type to the type of its target.

This allows downcasting a separate entity to a non-separate one, provided that the entity represents a non-separate object at runtime.

```
meet_friend (p: separate PERSON)
    do
        if attached {PERSON} p.friend as ap then
            visit (ap)
        end
    end
```

Genericity

- Entities of generic types may be separate

  list: LIST [BOOK]

  list: **separate** LIST [BOOK]

- Actual generic parameters may be separate

  list: LIST [**separate** BOOK]

  list: **separate** LIST [**separate** BOOK]

- All combinations are meaningful and useful
- Separateness is relative to object of generic class, e.g. elements of list: **separate** LIST [BOOK] are non-separate with respect to (w.r.t.) list but separate w.r.t. **Current**. Type combiners apply.