Concurrent Object-Oriented Programming

Prof. Dr. Bertrand Meyer

Lecture 7: SCOOP Type System
(based on work with Piotr Nienaltowski)
Refresher: Computational Model

- 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: Synchronization

Mutual exclusion:
- Locking through argument passing
- Routine body is critical section

Condition synchronization:
- Preconditions used as wait conditions

Re-synchronisation of client and supplier:
- Wait by necessity

Lock passing: through argument passing
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 can SCOOP do for us?

Beat enemy number one in concurrent world: atomicity violations

- Data races
- Illegal interleaving 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?
A traitor is an entity that

- Statically, is declared as non-separate
- During an execution, can become attached to a separate object.
-- in class C (client)
x1: separate T
a: A

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

-- supplier
class T feature
  b: A
end

Is this call valid?  
Yes

And this one?  
No
Traitors there...

-- in class C (client)
\(x_1: \text{separate } T\)
\(a: A\)

\(r(x: \text{separate } T)\)
\(\text{do}
\ x.f(a)
\text{end}\)

\(r(x_1)\)

-- supplier
\(\text{class T feature}\)
\(f(a: A)\)
\(\text{do}\)
\(a.f\)
\(\text{end}\)

And this one?

\(\times\)

Is this call valid?
\(\checkmark\)
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 be separate too.

```
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

...
Problem 1: unsoundness

expanded class E feature
    g: STRING
    f: INTEGER ...
end

x: E -- E is expanded.

consume_element (buffer: separate BUFFER [E])
    local
        s: STRING
    do
        x := buffer.item
        s := x·f·out -- Valid: call on expanded object.
        s := x·g·out -- Valid! call on separate reference.
    end

Traitor
Problem 2: limitations

expanded class \( E \) feature

\[
\begin{align*}
g &: \text{STRING} \\
f &: \text{INTEGER} \ldots
\end{align*}
\]

end

\( x_1: E \) \hspace{1cm} -- E is expanded.

\( y_1: \text{separate} \ Y \)

\( y_2: Y \)

\[
\begin{align*}
y_1 &= y_2 \\
y_1 \cdot r(x_1) &\quad -- \text{Invalid because } x_1 \text{ is not fully expanded}
\end{align*}
\]

\[
\begin{align*}
&\quad -- \text{In class } Y:\n
r(x: E) \\
do \ldots \end{align*}
\]
Problem 3: more limitations

class STRING feature

... 
  item alias "[ ]" (i: INTEGER): CHARACTER
    do  ... end 

append alias "+" (other: like Current)
  do  ... end 
end

-- in class T:

r (sl: separate LIST [STRING]) 
  do 
    sl • put (sl [1] + sl [2]) 
      -- Invalid but should be allowed
  end
Problem 4: even more limitations

```plaintext
r (l: separate LIST [STRING])
local
  s: separate STRING
do
  s := l[1]
  l.put (s) -- Invalid but should be allowed
end
```
Let’s make it better!

- **SCOOP rules**
  - Prevent almost all traitors, +
  - Are easy to understand by humans, +
  - Not sound, -
  - Too restrictive, -
  - No support for agents. -

- **Can we do better?**
  - Refine and formalize the rules
Using the type system

- How do you know whether an assignment or an argument passing are valid?
- The type system tells us!
A type system for SCOOP

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

Simplifies, refines and formalizes SCOOP rules

Integrates expanded types and agents with SCOOP

Tool for reasoning about concurrent programs
  ➢ May serve as basis for future extensions, e.g. for deadlock prevention schemes
Three components of a type

Notation:
\[ \Gamma |- x :: (\gamma, \alpha, C) \]

Under the binding \( \Gamma \),
\( x \) has the type \((\gamma, \alpha, C)\)

1. Attached/detachable: \( \gamma \in \{!, ?\} \)
   
   Some processor (top)
   \( x: \text{separate U} \)

2. Processor tag \( \alpha \in \{\bullet, T, \bot, <p>, <a\cdot handler>\} \)
   
   Current processor

3. Ordinary (class) type \( C \)
   
   No processor (bottom)
Examples

\[ u : U \quad \text{-- } \quad u :: (!, \bullet, U) \]
\[ v : \text{separate } V \quad \text{-- } \quad v :: (!, T, V) \]
\[ w : \text{detachable separate } W \quad \text{-- } \quad w :: (? , T, W) \]

-- Expanded types are attached and non-separate:

\[ i : \text{INTEGER} \quad \text{-- } \quad i :: (!, \bullet, \text{INTEGER}) \]
\[ \text{Void} \quad \text{-- } \quad \text{Void} :: (? , \bot, \text{NONE}) \]
\[ \text{Current} \quad \text{-- } \quad \text{Current} :: (!, \bullet, \text{Current}) \]
\[ x : \text{separate } \langle px \rangle T \quad \text{-- } \quad x :: (!, px, T) \]
\[ y : \text{separate } \langle px \rangle Y \quad \text{-- } \quad y :: (!, px, Y) \]
\[ z : \text{separate } \langle px \rangle Z \quad \text{-- } \quad z :: (!, px, Z) \]
Informal Subtyping Rules

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

\[ D \leq_{\text{Eiffel}} C \iff (\gamma, \alpha, D) \leq (\gamma, \alpha, C) \]

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:

\[ (\gamma, \bullet, C) \leq (\gamma, T, C) \]

\[ \bot \leq \text{any processor tag:} \]

\[ (\gamma, \bot, C) \leq (\gamma, \alpha, C) \]
So how does it help us?

We can rely on standard type rules
  • 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 $T$
    -- $a :: (!, T, T)$
b: $T$
    -- $b :: (!, \bullet, T)$
c: detachable $T$
    -- $c :: (? , \bullet, T)$
f ($x, y$: separate $T$) do ... end
    -- $x :: (!, T, T), y :: (!, T, T)$
g ($x$: $T$) do ... end
    -- $x :: (!, \bullet, T)$
h ($x$: detachable $T$): $<p> T$ do ... end
    -- $x :: (? , \bullet, T) : (!, p, T)$

f ($a, b$)
    Invalid
f ($a, c$)
    Invalid
g ($a$)
    Invalid
$a := h (b)$
    Invalid
$a := h (a)$
    Invalid
Is it always that simple?

Rules for feature calls are more complex

“Controllability” of target taken into account:
- Is target controlled?
- Is target’s handler accessible to client’s handler?

Type of formal arguments depends on type of target
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.
An expression \( \exp \) of type \((d, p, C)\) is **controlled** if and only if \( \exp \) is attached and satisfies one of the following conditions:

- \( \exp \) is non-separate, i.e. \( p = \bullet \)
- \( \exp \) appears in a routine \( r \) that has an attached formal argument \( a \) with the same handler as \( \exp \), i.e. \( p = a \cdot \text{handler} \)

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 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 (assume that Y and Z are descendants of T)

- **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.
Qualified explicit processor tags

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

\[ x: \text{separate} <\text{y.handler}> T \]

\[ \text{-- } x \text{ is handled by handler of } y \]

Attachment, object creation:

\[ r \text{ (list: separate LIST [T])} \]

\[ \text{local} \]

\[ s1, s2: \text{separate} <\text{list.handler}> \text{STRING} \]

\[ \text{-- } s1, s2 :: (!, <\text{list.handler}>, \text{STRING}) \]

\[ \text{do} \]

\[ s1 := \text{list [1]} \]
\[ s2 := \text{list [2]} \]
\[ \text{list.extend} (s1 + s2) \quad \text{-- Valid} \]
\[ \text{create } s1.\text{make_empty} \quad \text{-- } s1 \text{ created on list’s processor} \]
\[ \text{list.extend} (s1) \quad \text{-- valid} \]

\[ \text{end} \]
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:

- Formal arguments
- Result
Result type combinator

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

$T_{\text{result}} = T_{\text{target}} \times T_f$

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

$= (\alpha f, pr, TF)$

\[
\begin{array}{cccc}
\text{pf} & \bullet & T & \langle q \rangle \\
px & \bullet & T & T \\
& \bullet & T & T \\
& T & T & T \\
& \langle p \rangle & \langle p \rangle & T & T \\
\end{array}
\]
**Argument type combinator**

What is the expected actual argument type in \( \texttt{x.f(a)} \)?

\[
T_{\text{actual}} = T_{\text{target}} \otimes T_{\text{formal}}
\]

\[
= (\alpha x, px, TX) \otimes (\alpha f, pf, TF)
\]

\[
= (\alpha f, pa, D)
\]
Type combinators and expanded types

Expanded objects are always attached and non-separate. Both $\ast$ and $\otimes$ preserve expanded types

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

$x1 : T$  -- $x1 :: (\!, \bullet, T)$
$y1 : \text{separate} \ Y$  -- $y1 :: (\!, T, Y)$
$y1. r (x1)$
  -- $(\!, \bullet, T) \leq (\!, T, Y) \otimes (\!, \bullet, T)$
  -- so call is valid

expanded class
  T
feature
  g : STRING
  f : INTEGER
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)

\[ x_1: \text{separate } T \]
\[ a: A \]
\[ x :: (!, T, T) \]

\[ r (x: \text{separate } T) \]
\[ \text{do} \]
\[ a := x.a \]
\[ \text{end} \]

\[ r (x_1) \]
\[ a.f \]

-- supplier

\[ x_1 :: (!, T, T) \]

\[ \text{class } T \]

\[ \text{feature} \]
\[ a :: (!, \bullet, A) \]

\[ x :: (!, T, T) \]

\[ a: A \]

\[ \text{end} \]

\[ x.a :: (!, T, T) \times (!, \bullet, A) = (!, T, A) \]

\[ \text{not } (!, T, A) \not\in (!, \bullet, A) \]

so assignment is invalid

traitor
Recall: Traitors there...

-- in class C (client)

\[ x_1: \text{separate } T \]
\[ a: A \quad a :: (!, \bullet, A) \]
\[ x :: (!, T, T) \]
\[ r (x: \text{separate } T) \]
\[ \text{do} \]
\[ x.f (a) \]
\[ \text{end} \]
\[ r (x_1) \]

-- supplier

class T

feature

\[ f (a: A) \]
\[ \text{do} \]
\[ a.f \]
\[ \text{end} \]

\[ (!, \bullet, A) \leq (!, T, T) \otimes (!, \bullet, A) \]
\[ (!, \bullet, A) \not\leq (!, \perp, A) \]

so call is invalid
Implicit types

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

  $$x :: (!, T, T) = (!, x.\text{handler}, T)$$

  $$y :: (?\cdot, Y) \text{ no implicit type because } y \text{ is detachable}$$

  $$z :: (!, T, Z) \text{ no implicit type because } z \text{ is writable}$$

- $r (x: \text{separate } T; y: \text{detachable } Y)$

- local

- $z: \text{separate } Z$

- do ... end

- $s :: (!, \bullet, \text{STRING}) = (!, s.\text{handler}, \text{STRING})$

- $s: \text{STRING} = "I \text{ am a constant}"

- $u :: (!, T, U) = (!, u.\text{handler}, U)$

- $u: \text{separate } U \text{ once } ... \text{ end}$
meet_friend (person: separate PERSON)

local

  a_friend: PERSON

  do

    a_friend := person.friend -- invalid assignment.
    visit (a_friend)

  end
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.

```eiffel
meet_friend (person: separate PERSON)
  do
    if attached {PERSON} person.friend as a_friend then
      visit (a_friend)
    end
  end
end
```
Object creation

- **p**: PROCESSOR
  - Processor tag

- **a**: separate X
  - Create fresh processor for a
  - create a

- **b**: X
  - Place b on current processor
  - create b

- **c, d**: separate <p> X
  - Create fresh processor p for c
  - create c

- **create d**: Processor p already exists: place d on p
Without lock passing

\[ r \ (x: \text{separate } X; \ y: \text{separate } Y) \]

local

\[ z: \text{separate ANY} \]

\[ \text{do} \]

\[ x.f \]

\[ x.g (y) \]

\[ y.f \]

\[ z := x.\text{some_query} \]

\[ \text{end} \]
Lock passing

... 

\( x.f \)

\( x.g(y) \)

... 

\( y.f \)

\( g(y: \text{separate } Y) \)

\( \text{do} \)

\( y.f \)

... 

end
Lock passing

- Must not compromise the atomicity guarantees
- Clients must be able to decide to pass or not to pass a lock
- The mechanism should increase the expressiveness of the language, not restrict it
- The solution must be simple and well integrated with other language mechanism
Lock passing

- If a call \( x.f(a_1, \ldots, a_n) \) occurs in a routine \( r \) where some \( a_i \) is 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.

\[
\begin{align*}
\text{r (x: separate X; y: separate Y)} & \\
\text{local} & \\
\text{z: separate ANY} & \\
\text{do} & \\
\text{x.f} & \\
\text{x.g (y)} & \\
\text{y.f} & \\
\text{z := x.some_query} & \\
\text{end} & \\
\end{align*}
\]

Pass locks to \( g \) and wait for \( g \) to finish

Synchronous

Synchronous
<table>
<thead>
<tr>
<th>Actual</th>
<th>Formal</th>
<th>Attached</th>
<th>Detachable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference, controlled</td>
<td></td>
<td>Lock passing</td>
<td>no</td>
</tr>
<tr>
<td>Reference, uncontrolled</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Expanded</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>
Lock passing: example

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

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

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