Concurrent Object-Oriented Programming

Bertrand Meyer, Piotr Nienaltowski
Lecture 8 - 9:
Traitors, validity rules, type system
Outline

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

- 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
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

- 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?
Traits here...

-- 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

-- supplier

class X

feature
  a: A
end

Is this call valid?

TRAITOR! TRAITOR!

And this one?
-- 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

And this one?

Is this call valid?
Consistency rules – first attempt

- 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?
Separateness consistency rule (1)
If the source of an attachment (assignment instruction or argument passing) is separate, its target entity must be separate too.

\[ r \text{ (buffer: separate BUFFER [X]; x: X ) is} \]

- local
  \[ b1: \text{separate BUFFER [X]} \]
  \[ b2: \text{BUFFER [X]} \]
  \[ x2: \text{separate X} \]

- do
  \[ b1 := \text{buffer} \text{ -- valid} \]
  \[ b2 := b1 \text{ -- invalid} \]
  \[ r \text{ (b1, x2) \text{ -- invalid}} \]

end

Chair of Software Engineering

Piotr Nienaltowski, 23.05.2006, 30.05.2006
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
  ...

Chair of Software Engineering
Piotr Nienaltowski, 23.05.2006, 30.05.2006
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.

```plaintext
consume_element(buffer: separate BUFFER [X]) is
  local
    element: separate X
  do
    element := buffer.item
    . . .
  end
  -- in class BUFFER [G]
  item: G is
    . . .
```

Chair of Software Engineering
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.

\[
\text{store (buffer: separate BUFFER [X]; x: X) is}
\]
\[
\quad \text{do}
\]
\[
\quad \text{buffer.put (x) \quad -- X must be “fully expanded”}
\]
\[
\quad \text{end}
\]

\[
\quad \text{-- in class BUFFER [G]}
\]
\[
\text{put (element: G) is \quad -- G is not declared as separate anymore}
\]
\[
\ldots
\]
Problem 1: unsoundness

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

consume_element (buffer: separate BUFFER [X]) is

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

expanded class X

feature

  g: STRING

  f: INTEGER is . . .

end
Problem 2: limitations

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

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
Problem 4: even more limitations

\[ r \left( l : \text{separate LIST [STRING]} \right) \text{ is} \]

\begin{verbatim}
local
    s: separate STRING

do
    s := l.item (1)
    l.put (s)
end
\end{verbatim}

Invalid!!! But it should be allowed!
Let’s make it better!

- **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

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

- 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

- **Current processor**
  \[ x : X \]

- **Processor tag**
  \[ \alpha \in \{\bullet, T, \bot, <p>, <a.handler>\} \]

- **Attached/detachable**
  \[ \gamma \in \{!, \, Void\} \]

\[ \Gamma \vdash x :: (\gamma, \alpha, C) \]
### Examples

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

- **y**: `separate` $Y$  
  -- $y :: (!, \top, Y)$

- **z**: `? separate` $Z$  
  -- $z :: (?, \top, Z)$

- Expanded types are attached and non-separate  
  - **i**: `INTEGER`  
    -- $i :: (!, \bullet, \text{INTEGER})$

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

- **Current** is attached and non-separate  
  -- **Current** :: $(!, \bullet, C_{\text{Current}})$
Subtyping rules

- Since you don’t like Greek letters, I’ll keep it informal

- $\mathbb{T}_2 \leq \mathbb{T}_1$ means “$\mathbb{T}_2$ is a subtype of $\mathbb{T}_1$”

- Conformance on class types like in Eiffel, essentially based on inheritance
  \[
  D \preceq_{\text{Eiffel}} C \iff (\gamma, \alpha, D) \preceq (\gamma, \alpha, C)
  \]

- Attached $\leq$ detachable
  \[
  (\!, \alpha, C) \preceq (\?, \alpha, C)
  \]

- Any processor tag $\leq T$
  \[
  (\gamma, \alpha, C) \preceq (\gamma, T, C)
  \]

- In particular, non-separate $\leq T$
  \[
  (\gamma, \cdot, C) \preceq (\gamma, T, C)
  \]

- $\perp \leq$ any processor tag
  \[
  (\gamma, \perp, C) \preceq (\gamma, \alpha, C)
  \]
So how does it help us?

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

\[
\begin{align*}
\Gamma |- x :: \tau_x, & \quad \Gamma |- e :: \tau_e, \quad \Gamma |- \tau_e \leq \tau_x \\
\text{[Assign]} & \\
\Gamma |- x := e
\end{align*}
\]

- No need for special validity rules for separate
Is it always that simple?

- 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

- 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 \( f \).

\[
\begin{align*}
\Gamma \vdash e :: TT_e, & \quad \Gamma \vdash \text{isTargettable}(TT_e), & \quad \Gamma \vdash \text{ClassType}(TT_e) = C \\
\Gamma \vdash \text{FeatureType}(C, f) = TT'_1 \times \cdots \times TT'_n \rightarrow TT_{res} & \quad n \geq 0 \\
\Gamma \vdash a_i :: TT_i & \quad i \in 1..n, & \quad \bar{a} = a_1..a_n, & \quad \Gamma \vdash \forall i \in 1..n \quad TT_i \preceq TT_e \otimes TT'_i
\end{align*}
\]

\( \Gamma \vdash e.f(\bar{a}) :: TT_e \ast TT_{res} \)  \hspace{1cm} (T-QCallQual)

- **Type combinators** necessary to calculate relative type
  - formal arguments \( \otimes \)
  - result \( \ast \)
Type combinator \(*\) (result type)

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

<table>
<thead>
<tr>
<th></th>
<th>(\lambda)</th>
<th>(\beta)</th>
<th>(\alpha)</th>
<th>(T)</th>
<th>(\langle q\rangle)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\langle p\rangle)</td>
<td>(\langle p\rangle)</td>
<td>(T)</td>
<td>(T)</td>
<td>(T)</td>
<td>(T)</td>
</tr>
</tbody>
</table>
Type combinator $\otimes$ (formals)

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

<table>
<thead>
<tr>
<th></th>
<th>$\beta$</th>
<th>$\alpha$</th>
<th>$T$</th>
<th>$\langle q \rangle$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha$</td>
<td></td>
<td>$\cdot$</td>
<td>$T$</td>
<td>$\langle q \rangle$</td>
</tr>
<tr>
<td>$\cdot$</td>
<td>$\cdot$</td>
<td></td>
<td>$T$</td>
<td>$\perp$</td>
</tr>
<tr>
<td>$\cdot$</td>
<td>$\cdot$</td>
<td></td>
<td>$T$</td>
<td>$\perp$</td>
</tr>
<tr>
<td>$T$</td>
<td>$\perp$</td>
<td></td>
<td>$T$</td>
<td>$\perp$</td>
</tr>
<tr>
<td>$\langle p \rangle$</td>
<td>$\langle p \rangle$</td>
<td>$T$</td>
<td>$\perp$</td>
<td></td>
</tr>
</tbody>
</table>
Expanded types

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

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

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

\[
\begin{align*}
\text{my}_x &\colon X & \quad \text{-- my}_x &\colon (!, \bullet, X) \\
\text{my}_y &\colon \text{separate} \ Y & \quad \text{-- my}_y &\colon (!, \top, Y) \\
\text{...} &\quad \text{--} & \quad \text{\hspace{10cm} expanded class} X \\
\text{my}_y.r (\text{my}_x) &\quad \text{--} & \quad \text{\hspace{10cm} feature} \\
\text{--} & \quad \text{\hspace{10cm} g: STRING} \\
\text{--} & \quad \text{\hspace{10cm} f: INTEGER is \ldots} \\
\text{--} & \quad \text{\hspace{10cm} end} \\
\text{--} & \quad \text{\hspace{10cm} so call is valid}
\end{align*}
\]
 Expanded types: deep_import

- 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...

-- in class C (client)

\[
\begin{align*}
my_x &: separate X \\
\text{a} &: A \\
\end{align*}
\]

...  

\[
\begin{align*}
\text{r} (x &: separate X) \text{ is} \\
& \text{do} \\
& \quad a := x.a \\
& \text{end} \\
\end{align*}
\]

...  

\[
\begin{align*}
\text{r} (\text{my}_x) \\
\text{a.f}
\end{align*}
\]

-- supplier

\[
\begin{align*}
class X \\
\text{feature} \\
\quad a &: A \\
\text{end}
\end{align*}
\]

\textbf{traitor! traitor!}
Type system eliminates them

-- in class C (client)
my_\textit{x}: \textbf{separate} \, \textit{X}
    -- my_\textit{x} :: (!, \top, \textit{X})
a: A    -- a :: (!, \bullet, \textit{X})
...

\textit{r (x: separate X) is}    -- x :: (!, \top, \textit{X})
    \textbf{do}
        a := x.a    -- x.a :: (!, \top, \textit{X}) \times (!, \bullet, A) = (!, \top, A)

    -- (!, \top, A) \notin (!, \bullet, A)

    -- therefore assignment is invalid

end
...

\textit{r (my_x)}    -- (!, \top, \textit{X}) \leq (!, \bullet, C) \otimes (!, \top, \textit{X}) , so call is valid
a.f

\textit{Class X}
\textbf{feature}
a: A    -- a :: (!, \bullet, A)
end
Recall: traitors there...

-- in class C (client)

\[
\begin{align*}
my_x & : \text{separate } X \\
a & : A \\
\end{align*}
\]

\[
\begin{align*}
\text{traitor! traitor!} \\
r (x : \text{separate } X) \text{ is} \\
\text{do} \\
x.f (a) \\
\text{end} \\
\end{align*}
\]

\[
\begin{align*}
r (my_x) \\
\end{align*}
\]

-- supplier

\[
\begin{align*}
\text{class } X \\
\text{feature} \\
f (a : A) \text{ is} \\
\text{do} \\
a.f \\
\text{end} \\
\end{align*}
\]

\[
\begin{align*}
\text{end} \\
\end{align*}
\]
Type system eliminates them

-- in class C (client)

my_\_x: separate X

-- my_\_x :: (!, \tau, X)

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

...

r (x: separate X) is -- x :: (!, \tau, X)

do

x.f (a)

-- (!, \bullet, A) \not\subseteq (!, \tau, X) \otimes (!, \bullet, A)

-- (!, \bullet, A) \not\subseteq (!, \bot, A)

-- therefore call is invalid

end

...

r (my_\_x) (!, \tau, X) \leq (!, \bullet, C) \otimes (!, \tau, X) , so call is valid

-- supplier

class X

feature

f (a: A) is

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

do

a.f

end

end

Chair of Software Engineering

Piotr Nienaltowski, 23.05.2006, 30.05.2006
Explicit processor tags

\[
p: \text{HANDLER} \quad \text{-- Tag declaration}
\]
\[
x: \text{separate } \langle p \rangle \quad X \quad \text{-- } x :: (!, \langle p \rangle, X)
\]
\[
y: \text{separate } \langle p \rangle \quad Y \quad \text{-- } y :: (!, \langle p \rangle, Y)
\]
\[
z: \text{separate } Z \quad \text{-- } z :: (!, T, Z)
\]

- **Attachment**
  - Assume that Y and Z inherit from X
  \[
x := y \quad \text{-- Valid because (!, \langle p \rangle, Y) } \leq \ (!, \langle p \rangle, X)
\]
  \[
y := z \quad \text{-- Invalid because (!, T, Z) } \leq \ (!, \langle p \rangle, Y)
\]

- **Object creation**
  \[
\text{create } x \quad \text{-- Fresh processor is created to handle } x.
\]
  \[
\text{create } y \quad \text{-- No new processors created; } y \text{ is put on } x\text{’s processor.}
\]
Implicit processor tags

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

  \[ x: \text{separate} \ <\text{roe.handler}> \ X \]
  -- x is handled by roe’s handler.

- Attachment, object creation

  \[ r (\text{list}: \text{separate} \ \text{LIST} [X]) \]
  \hspace{1cm} \text{local}
  \hspace{1cm} s1, s2: \text{separate} \ <\text{list.handler}> \ STRING
  -- s1 :: (!, <list.handler>, STRING), s2 alike
  \hspace{1cm} \text{do}
  \hspace{1.5cm} s1 := \text{list.item} (1)
  \hspace{1.5cm} s2 := \text{list.item} (2)
  \hspace{1.5cm} \text{list.extend} (s1 + s2) \hspace{0.5cm} -- \text{Valid}
  \hspace{1.5cm} \text{create} \ s1.make_empty \hspace{0.5cm} -- \text{s1 created on list’s processor}
  \hspace{1.5cm} \text{list.extend} (s1) \hspace{0.5cm} -- \text{Valid}
  \hspace{1cm} \text{end} \]
meet_friend (person: separate PERSON) is

local

  a_friend: PERSON

do

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

end
Handling false traitors

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

Chair of Software Engineering

Piotr Nienaltowski, 23.05.2006, 30.05.2006
Assignment attempt

- Like in Eiffel but also “downcasts” processor tags
  - “deep downcast” over expanded attributes

- Eiffel standard has replaced ?= with object test

```plaintext
if {a_friend: PERSON} person.friend then
  visit (a_friend)
end
```
That’s all, folks!

Questions?