Lecture 2 (8):
Validity rules, type system
Outline

- Refresher on Lecture 1
- Problem of traits
- Validity rules – first attempt
- The need for a type system
- Examples
Summary: 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**
Software system

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

<o1> denotes o1’s owner
<o1> = P1
Summary: 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
Summary: separate argument rule

The target of a separate call must be a formal argument of the enclosing routine

Separate call: $a.f(...)$ where $a$ is a separate entity
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 should do for us

- Beat *enemy number one* in concurrent world, i.e. data races
  - *(traditional)* *Data race occurs when two or more threads concurrently access a shared variable, and at least one is writing.*

→ *(SCOOP)* *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.
  - *Kill ‘em all!*
-- in class C (client)
x: separate X
a: A
...
\( r(\text{an}_x: \text{separate } X) \) is
    do
        a := an_x.a
    end
...
\( r(x) \)
\( a.f \)

-- supplier
class X
feature
    a: A
end

Is this call valid?

TRAITOR! TRAITOR!

And this one?
Consistency rules – first attempt

- Four consistency rules
- Should prevent data races
  - eliminate traitors
- Written in English
- Easy to understand by programmers
**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
```

*Chair of Software Engineering*  
Piotr Nienaltowski, 17.05.2005
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
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
    ...
```
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 with expanded types

\[ x: \text{X} \quad \text{-- \ X is expanded, see below.} \]

\textit{consume_element} (buffer: \textbf{separate BUFFER [X]} \textbf{is})

\textbf{local}

\[ s: \text{STRING} \]

\textbf{do}

\[ x := \text{buffer.item} \]

\[ s := x.f.out \quad \text{-- Valid: call on expanded object.} \]

\[ s := x.g.out \quad \text{-- Valid! call on separate reference.} \]

\textbf{end}

\textbf{expanded class X}

\textbf{feature}

\[ g: \text{STRING} \]

\[ f: \text{INTEGER} \textbf{is} \ldots \]

\textbf{end}
Separateness consistency rule (5)
If the source of an attachment is the result of a separate call to a function returning an expanded type, the class that defines this type may not include, directly or indirectly, any non-separate attribute of a reference type.

- Handling of expanded objects still unsatisfactory
- Many useful programs ruled out
- Is it the only missing rule?
Is that the right solution?

- SCOOP 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!
The question for today

How do you know whether assignment

\[ x := y \]

-- \( x \) and \( y \) are declared as

-- \( x: \text{CLASS}_X; y: \text{CLASS}_Y \)

and argument passing

\[ r (x) \]

-- \( r \) is declared as \( r (an_x: \text{SOME}_\text{CLASS}) \)

are valid?
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

- 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)
Examples