Concepts of Concurrent Computation

Bertrand Meyer
Sebastian Nanz

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

- Software systems are 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
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? ✔

And this one? ✗
Traitors there...

-- In class C (client)
\( x1: \text{separate } T \)
\( a: A \)

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

-- Supplier
class T feature
\( f(b: A) \)
\[ \text{do} \]
\[ b.f \]
\[ \text{end} \]

\text{Is this call valid?} \quad \checkmark

\text{And this one?} \quad \times
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.

```haskell
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: too restrictive

```plaintext
r (l: separate LIST [STRING])
local
  s: separate STRING
do
  s := l [1]
  l.put (s)  -- Invalid according to Rule 2
  -- but should be allowed
end
```
Let’s make it better!

- **SCOOP 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
Using the type system

- How do we know whether an assignment or an argument passing are valid?
- We use a type system
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
Three components of a type

Notation:

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

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

2. Processor tag \( \alpha \in \{., T, \perp, \langle p\rangle, \langle a\cdot handler\rangle\} \)

3. Ordinary (class) type \( C \)

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

Some processor (top)
\( x: \text{separate } U \)

Current processor

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)
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 \( \leq T \):

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

\( \bot \leq \) any processor tag:

\[ (\gamma, \bot, C) \leq (\gamma, \alpha, C) \]

Standard Eiffel (non-SCOOP) conformance
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 \( C \)  
\( \quad \text{-- a : (!, T, C)} \)
b: \( C \)  
\( \quad \text{-- b : (!, \bullet, C)} \)
c: detachable \( C \)  
\( \quad \text{-- c : (? , \bullet, C)} \)
f \( (x, y: \text{separate } C) \) do ... end  
\( \quad \text{-- x : (!, T, C), y : (!, T, C)} \)
g \( (x: C) \) do ... end  
\( \quad \text{-- x : (!, \bullet, C)} \)
h \( (x: \text{detachable } C): \text{separate } <p> C \) do ... end  
\( \quad \text{-- x : (? , \bullet, C) : (!, p, C)} \)

f \((a, b)\)  
\( \quad \text{Valid} \)
f \((a, c)\)  
\( \quad \text{Invalid} \)
g \((a)\)  
\( \quad \text{Invalid} \)
a := h \((b)\)  
\( \quad \text{Valid} \)
a := h \((a)\)  
\( \quad \text{Invalid} \)
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 \(\text{exp} \) of type \((d, p, C)\) is **controlled** if and only if \(\text{exp} \) is attached and satisfies _any_ of the following conditions:

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

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.

- \( p: \text{PROCESSOR} \)  -- Tag declaration
- \( x: \text{separate } <p> \ T \)  -- \( x: (!, <p>, T) \)
- \( y: \text{separate } <p> \ Y \)  -- \( y: (!, <p>, Y) \)
- \( z: \text{separate } \ Z \)  -- \( z: (!, T, Z) \)

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

- \( x := y \)  -- Valid because \( (!, <p>, Y) \leq (!, <p>, T) \)
- \( y := z \)  -- Invalid because \( (!, T, Z) \nleq (!, <p>, Y) \)

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

\( p: \text{PROCESSOR} \)

\( a: \text{separate} \ X \)
\( b: \ X \)
\( c, d: \text{separate} \ <p> \ X \)

create \( a \)

Create fresh processor for \( a \)

create \( b \)

Place \( b \) on current processor

create \( c \)

Create fresh processor \( p \) for \( c \)

create \( d \)

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)

\[ x: \text{separate} \langle y.\text{handler}\rangle \ T \]
\[ -- x \text{ is handled by handler of } y \]

Attachment, object creation:
\[ \text{r (list: separate LIST [T])} \]
\[ \text{local} \]
\[ s1, s2: \text{separate} \langle \text{list.handler}\rangle \ \text{STRING} \]
\[ -- s1, s2: (!, \langle \text{list.handler}\rangle, \text{STRING}) \]
\[ \text{do} \]
\[ s1 := \text{list [1]} \]
\[ s2 := \text{list [2]} \]
\[ \text{list.extend (s1 + s2)} \quad -- \text{Valid} \]
\[ \text{create s1.make_empty} \quad -- \text{s1 created on list’s processor} \]
\[ \text{list.extend (s1)} \quad -- \text{Valid} \]
\[ \text{end} \]
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 $x \cdot f (...)$?

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

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

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

<table>
<thead>
<tr>
<th>px</th>
<th>pf</th>
<th>T</th>
<th>$&lt;q&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>$&lt;p&gt;$</td>
<td>$&lt;p&gt;$</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>
What is the expected actual argument type in $x.f\,(a)$?

$$T_{actual} = T_{target} \otimes T_{formal}$$

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

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

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

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

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

x1: T  --  x1 : (!, \bullet, T)

y1: separate Y  --  y1 : (!, T, Y)

y1. \text{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)

\[
\begin{align*}
x_1 & \colon \text{separate } T \\
a & \colon A
\end{align*}
\]

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

\[
\begin{align*}
r (x_1) & \\
a.f & \text{Traitor}
\end{align*}
\]

-- Supplier class $T$

\[
\begin{align*}
x_1 & \colon (!, T, T) \\
a & \colon (!, \bullet, A)
\end{align*}
\]

\[
\begin{align*}
\text{feature } a & \colon A \\
\text{end}
\end{align*}
\]

\[
\begin{align*}
x.a & \colon (!, T, T) \times (!, \bullet, A) = (!, T, A) \\
(!, T, A) & \lesssim (!, \bullet, A)
\end{align*}
\]

So assignment is invalid
Recall: 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
\[ x_1: (!, T, T) \]
\[ a: (!, \bullet, A) \]
\[ f(a: A) \]
\[ \text{do} \]
\[ a.f \]
\[ \text{end} \]

\[ (!, \bullet, A) \preceq (!, T, T) \otimes (!, \bullet, A) \]
\[ (!, \bullet, A) \preceq (!, \bot, A) \]

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

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

local

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

\[ z :: (!, T, Z) \] no implicit type because $z$ is writable

\[ s :: (!, \cdot, \text{STRING}) = (!, s.\text{handler}, \text{STRING}) \]

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

\[ u :: (!, T, U) = (!, u.\text{handler}, U) \]

$u: \text{separate } U \ \text{once } ... \ \text{end}$
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.

```eiffel
meet_friend (p: separate PERSON)
  do
    if attached {PERSON} p.friend as ap then
      visit (ap)
    end
  end
end
```
The need for lock passing

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

local

\( z: \text{separate } \text{ANY} \)

do

\( x.f \)
\( x.g \ (y) \)
\( y.f \)
\( z := x.\text{some_query} \)

end
Lock passing

... x.f
x.g (y)
y.f

Pc

Py

Px

x.f
x.g (y)
g (y: separate Y)
do
... y.f
... y.f
... end
Lock passing

• Must not compromise the atomicity guarantees
• Clients must control whether 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 one or more \( a_i \) are 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.

```plaintext
r (x: separate X; y: separate Y)

local
    z: separate ANY

do
    x.f
    x.g(y)
    y.f
    z := x.some_query
end
```

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

Synchronous

Synchronous
# Lock passing combinations

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

<table>
<thead>
<tr>
<th>Class C Feature</th>
<th>Class X Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1: X</td>
<td>f (i: INTEGER) do ... end</td>
</tr>
<tr>
<td>z1: separate Z</td>
<td>g (a: separate ANY) do ... end</td>
</tr>
<tr>
<td>c1: separate C</td>
<td>h (c: separate C): INTEGER do c.p (...) end</td>
</tr>
<tr>
<td>i: INTEGER</td>
<td>m (a: detachable separate ANY) do ... end</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>r (x: separate X; y: separate Y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>do</td>
</tr>
<tr>
<td>x1.f (5)</td>
</tr>
<tr>
<td>x1.g (x)</td>
</tr>
<tr>
<td>i := x1.h (Current)</td>
</tr>
<tr>
<td>x.f (10)</td>
</tr>
<tr>
<td>x.g (z1)</td>
</tr>
<tr>
<td>x.g (y)</td>
</tr>
<tr>
<td>x.m (y)</td>
</tr>
<tr>
<td>i := x.h (c1)</td>
</tr>
<tr>
<td>i := x.h (Current)</td>
</tr>
</tbody>
</table>

Non-separate, no wait by necessity, no lock passing
Non-separate, no wait by necessity, lock passing (vacuous)
Non-separate, wait by necessity, lock passing (vacuous)
Separate, no wait by necessity, no lock passing
Separate, no wait by necessity, no lock passing
Separate, no wait by necessity, lock passing
Separate, no wait by necessity, no lock passing
Separate, wait by necessity, no lock passing
Separate, wait by necessity, lock passing