Concepts of Concurrent Computation

Bertrand Meyer
Sebastian Nanz

Lecture 6: SCOOP type system
A traitor is an entity that
- Statically, is declared as non-separate
- During an execution, can become attached to a separate object
--- In class \texttt{C (client)}

\texttt{x1: separate T}

\texttt{a: A}

\texttt{r (x: separate T)}

\hspace{1em} \texttt{do}

\hspace{2em} \texttt{a := x.b}

\hspace{1em} \texttt{end}

--- Supplier

\texttt{class T feature}

\hspace{1em} \texttt{b: A}

\texttt{end}

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

\texttt{r (x1)}

\texttt{a.f}

\texttt{And this one?} \quad \xmark
-- In class C (client)
x1: separate T
a: A

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

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

And this one?

Is this call valid?
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.

```plaintext
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

```plaintext
r (l: separate LIST [STRING])
    local
        s: separate STRING
    do
        s := l[1]
        l.extend (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, -

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

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

2. Processor tag \( \alpha \in \{\text{.}, \top, \bot, <p>, <a\cdot\text{handler}>\} \)

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)
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:

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

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

\[ (\gamma, \perp, C) \leq (\gamma, \alpha, C) \]
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
b: C
c: detachable C
df (x, y: separate C) do ... end
eg (x: C) do ... end
h (x: detachable C): separate <p> C
  do ... end

f (a, b) ✓
f (a, c) Invalid
eg (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 \( \text{exp} \) 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: 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
Qualified explicit processor tags

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

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

-- \( x \) is handled by handler of \( y \)

Attachment, object creation:
\[ r \ (\text{list: separate LIST [T]}) \]
local
\[ s1, s2: \text{separate } <\text{list.handler}> \text{ STRING} \]

-- \( s1, s2 : (!, <\text{list.handler}>, \text{STRING}) \)
do
\[ s1 := \text{list [1]} \]
\[ s2 := \text{list [2]} \]
\[ \text{list.extend } (s1 + s2) \]

-- Valid
\[ \text{create } s1.\text{make_empty} \]
-- \( s1 \) created on list’s processor
\[ \text{list.extend } (s1) \]

-- Valid
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 $T_{\text{result}}$ of a query call $x \cdot f (...)$?

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

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

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

<table>
<thead>
<tr>
<th></th>
<th>$pf$</th>
<th>$\cdot$</th>
<th>$T$</th>
<th>$&lt;q&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$px$</td>
<td>$\cdot$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\cdot$</td>
<td>$\cdot$</td>
<td>$T$</td>
<td>$T$</td>
<td></td>
</tr>
<tr>
<td>$T$</td>
<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>
<td></td>
</tr>
</tbody>
</table>
Argument type combinator

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

\[
T_{\text{actual}} = T_{\text{target}} \otimes T_{\text{formal}} = (\alpha \mathbf{x}, \mathbf{p} \mathbf{x}, \mathbf{T} \mathbf{x}) \otimes (\alpha \mathbf{f}, \mathbf{p} \mathbf{f}, \mathbf{T} \mathbf{f}) = (\alpha \mathbf{f}, \mathbf{p} \mathbf{a}, \mathbf{T} \mathbf{f})
\]

\[
\begin{array}{c|c|c|c}
\mathbf{p} \mathbf{x} & \mathbf{p} \mathbf{f} & \mathbf{T} & <q> \\
\hline
\mathbf{p} \mathbf{x} & . & \mathbf{T} & <q> \\
\hline
. & . & \mathbf{T} & \bot \\
\hline
\mathbf{T} & \bot & \mathbf{T} & \bot \\
\hline
<q> & <q> & \mathbf{T} & \bot
\end{array}
\]
Type combinators and expanded types

Expanded objects are always attached and non-separate. Both * and ⊗ preserve expanded types

- \((\gamma, \alpha, C) * (!, \cdot, \text{INTEGER}) = (!, \cdot, \text{INTEGER})\)
- \((\gamma, \alpha, C) \otimes (!, \cdot, \text{BOOLEAN}) = (!, \cdot, \text{BOOLEAN})\)

\[\begin{align*}
x1 &: \text{EXP} & \text{-- } x1 &: (!, \cdot, \text{EXP}) \\
y1 &: \text{separate}\ Y & \text{-- } y1 &: (!, T, Y) \\
y1 \cdot r (x1) & \text{-- } (!, \cdot, \text{EXP}) \leq (!, T, Y) \otimes (!, \cdot, \text{EXP}) \\
& \text{-- so call is valid}\end{align*}\]

expanded class EXP

feature

end

\[\begin{align*}r (a: \text{EXP}) & \text{ do ... end}\end{align*}\]
Recall: Traitors here...

-- in class C (client)

\[ x1: \text{separate } T \]
\[ a: A \]
\[ r(x: \text{separate } T) \]
\[ \text{do} \]
\[ a := x.b \]
\[ \text{end} \]

-- Supplier class T

\[ x1: (!, T, T) \]
\[ b : (!, \cdot, A) \]
\[ \text{end} \]

So assignment is invalid
Recall: Traitors there...

-- in class C (client)
\[ x_1: \text{separate } Z \]
\[ b: A \]
\[ r(x: \text{separate } Z) \]
\[ \text{do} \]
\[ x.f(b) \]
\[ \text{end} \]

\[ r(x_1) \]

-- supplier
\[ x_1: (!, T, Z) \]
\[ b: (!, \cdot, A) \]
\[ a: (!, \cdot, A) \]
\[ f(a: A) \]
\[ \text{do} \]
\[ a.f \]
\[ \text{end} \]

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

So call is invalid
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.

\[
\text{meet\_friend (p: separate PERSON)}
\]

\[
\begin{align*}
\text{do} \\
\quad \text{if attached \{PERSON\} p.friend as ap then} \\
\quad \quad \text{visit (ap)} \\
\quad \text{end} \\
\text{end}
\end{align*}
\]
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.