Concepts of Concurrent Computation

Bertrand Meyer
Sebastian Nanz
Chris Poskitt

Lecture 7: 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 C (client)
x1: separate T
a: A

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

-- Supplier
class T feature
  b: A
end

Is this call valid? ✓

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

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

\(r (x1)\)

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

Is this call valid?

And this one?

\(\times\)

\(\times\)
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 also be separate.

```
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 \( \alpha \in \{., T, \bot, <p>, <a\cdot handler>\} \)

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 $\leq T$:

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

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

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

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

f (a, b) ✓
f (a, c) Invalid
f (a, c) Invalid
g (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. \( p = a \cdot \text{handler} \)

A call \( x \cdot f(a) \) 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

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

Processor tag

Create fresh processor for a

Place b on current processor

Create fresh processor p for c

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)

\[ x: \text{separate} \langle y.\text{handler}\rangle \ T \]

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

Attachment, object creation:

\[
\begin{align*}
    r (\text{list: separate LIST [T]}) \\
    \text{local} \\
    \quad s1, s2: \text{separate} \langle \text{list.handler}\rangle \ \text{STRING} \\
    \quad \quad \quad \quad \quad \text{-- s1, s2 : (!, \langle \text{list.handler}\rangle, \text{STRING})} \\
    \text{do} \\
    \quad \text{s1 := list [1]} \\
    \quad \text{s2 := list [2]} \\
    \quad \text{list.extend (s1 + s2)} \quad \text{-- Valid} \\
    \quad \text{create s1.make_empty} \quad \text{-- s1 created on list's processor} \\
    \quad \text{list.extend (s1)} \quad \text{-- Valid} \\
\end{align*}
\]

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 of:

- Formal arguments
- Result
What is the type $T_{\text{result}}$ of a query call $x \cdot f (...)$?

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

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

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

\[
\begin{array}{c|c|c|c}
\text{px} & \text{pf} & \text{T} & \langle q \rangle \\
\hline
\text{px} & \cdot & \text{T} & \langle q \rangle \\
\hline
\cdot & \cdot & \text{T} & \text{T} \\
\hline
\text{T} & \text{T} & \text{T} & \text{T} \\
\hline
\langle p \rangle & \langle p \rangle & \text{T} & \text{T} \\
\end{array}
\]
**Argument type combinator**

What is the expected actual argument type in $x.f(a)$?

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

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

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

<table>
<thead>
<tr>
<th></th>
<th>pf</th>
<th></th>
<th>T</th>
<th>$&lt;q&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>px</td>
<td>.</td>
<td></td>
<td>T</td>
<td>$&lt;p&gt;$</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>.</td>
<td>T</td>
<td>$\bot$</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
<td>T</td>
<td>$\bot$</td>
</tr>
<tr>
<td>T</td>
<td>$\bot$</td>
<td>T</td>
<td>$\bot$</td>
<td></td>
</tr>
<tr>
<td>$&lt;p&gt;$</td>
<td>$&lt;p&gt;$</td>
<td>T</td>
<td>$\bot$</td>
<td></td>
</tr>
</tbody>
</table>
Type combinators and expanded types

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

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

x1: EXP    -- x1 : (!, ., EXP)
y1: separate Y    -- y1 : (!, T, Y)
y1.r (x1)    -- (!, ., EXP) ≤ (!, T, Y) ⊗ (!, ., EXP)
    -- so call is valid

r (a: EXP) do ... end

expanded class EXP
feature
    g: STRING
    f: INTEGER
end
Recall: Traitors here...

-- in class C (client)

\[ x_1: \text{separate} \ T \]
\[ a: A \]

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

\[ r(x_1) \]

\[ a.f \]

-- Supplier class T

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

\[ b: (!, \bullet, A) \]

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

\[ \text{end} \]

\[ \text{So assignment is invalid} \]
Recall: Traitors there...

-- in class C (client)
\[ x1: \text{separate } Z \]
\[ b: A \]
\[ r (x: \text{separate } Z) \]
\[ \text{do} \]
\[ x.f (b) \]
\[ \text{end} \]
\[ r (x1) \]

-- supplier

\[ x1 : (!, T, Z) \]
\[ a : (!, \cdot, A) \]
\[ \text{feature} \]
\[ f (a: A) \]
\[ \text{do} \]
\[ a.f \]
\[ \text{end} \]

So call is invalid

\((!, \cdot, A) \leq (!, T, Z) \otimes (!, \cdot, A) \)
\((!, \cdot, A) \leq (!, \bot, A) \)
meet_friend (p: separate PERSON)
local
    a_friend: PERSON
    do
        a_friend := p.friend  -- Invalid
        visit (a_friend)
    end
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
```
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.