Concepts of Concurrent Computation

Bertrand Meyer
Sebastian Nanz

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
Traitors here...

-- In class C (client)
x1: separate T
a: A

-- Supplier
class T feature
b: A
end

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

Is this call valid?  ✔

r (x1)
a.f

And this one?  ❌
Traitors there...

-- In class C (client)
\[
x_1: \text{separate } T \\
a: A
\]
\[
r (x: \text{separate } T) \\
do \\
x.f (a) \\
end
\]
\[
r (x_1)
\]

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

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.put (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, -
- No support for agents -

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

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

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

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

\[ u: U \quad \text{-- } u : (!, \bullet, U) \]
\[ v: \text{separate } V \quad \text{-- } v : (!, T, V) \]
\[ w: \text{detachable separate } W \quad \text{-- } w : (?, T, W) \]

\text{-- Expanded types are attached and non-separate:}
\[ i: \text{INTEGER} \quad \text{-- } i : (!, \bullet, \text{INTEGER}) \]
\[ \text{Void} \quad \text{-- } \text{Void} : (?, \perp, \text{NONE}) \]
\[ \text{Current} \quad \text{-- } \text{Current} : (!, \bullet, \text{Current}) \]
\[ x: \text{separate } <px> T \quad \text{-- } x : (!, px, T) \]
\[ y: \text{separate } <px> Y \quad \text{-- } y : (!, px, Y) \]
\[ z: \text{separate } <px> Z \quad \text{-- } z : (!, px, Z) \]
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 ≤ detachable:

\[ (!, \alpha, C) \leq (? , \alpha, C) \]

Any processor tag ≤ T:

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

In particular, non-separate ≤ T:

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

⊥ ≤ 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
f (a, c)  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 $exp$ of type $(d, p, C)$ is **controlled** if and only if $exp$ is attached and satisfies *any* of the following conditions:

- $exp$ is non-separate, i.e. $p = \bullet$
- $exp$ appears in a routine $r$ that has an attached formal argument $a$ with the same handler as $exp$, 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 rely on a processor attribute.

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

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 \( (!, T, Z) \not\leq (!, <p>, Y) \)

Object creation

- \( \text{create} \ x \) -- Fresh processor created to handle \( x \).
- \( \text{create} \ y \) -- 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)

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

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

- Formal arguments
- Result
Result type combinator

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

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

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

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

```
x \times f
```
Argument type combinator

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

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

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

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

$x \otimes f$
Type combinators and expanded types

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

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

$x1: \text{EXP} \quad -- \quad x1 : (\!, \!, \text{EXP})$

$y1: \text{separate} \ Y \quad -- \quad y1 : (\!, \ T, \ Y)$

$y1 \cdot r (x1) \quad -- \quad (\!, \ T, \ Y) \otimes (\!, \!, \text{EXP})$
- so call is valid

expanded class
Exp
feature
g: \text{STRING}
f: \text{INTEGER}
end

r (a: \text{EXP}) do ... 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)
\[
x_1: \text{separate} \quad T
\]
\[
a: \ A
\]
\[
r (x: \text{separate} \quad T)
\]
\[
do
\]
\[
a := x.a
\]
\[
end
\]
\[
r (x1)
\]
\[
a.f
\]

-- Supplier

class T

feature

\[
a : (!, \bullet, A)
\]
\[
x1 : (!, \top, T)
\]
\[
ar (x1)
\]

So assignment is invalid
Recall: Traitors there...

-- in class C (client)

\[ x_1: \text{separate } Z \]

\[ b: A \]

\[ r (x: \text{separate } Z) \]

\[ \text{do} \]

\[ x.f (b) \]

\[ \text{end} \]

\[ r (x_1) \]

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

So call is invalid

-- supplier

class Z

feature

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

\[ f (a: A) \]

\[ \text{do} \]

\[ a.f \]

\[ \text{end} \]

end
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

```plaintext
r (x: separate T; y: detachable Y)
local
z: separate Z
do ... end
```

$x :: (!, T, T) = (!, x.\text{handler}, T)$

$y :: (? , \bullet , Y)$ no implicit type because $y$ is detachable

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

$s :: (!, \bullet , \text{STRING}) = (!, s.\text{handler}, \text{STRING})$

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

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

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

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)
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.