SCOOP
Simple Concurrent Object-Oriented Programming
Lecture 3 (9):
Type system for SCOOP
Outline

- Refresher on Lecture 2
- Type system for SCOOP
- Validity rules – second attempt
- Examples
- Handling false traitors
Software system

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

We also say “P1 is o1’s owner”
What SCOOP should do for us

- Beat *enemy number one* in concurrent world, i.e. data races
  - *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.
--- in class C (client)
x: separate X
a: A
...
\[ \textit{r (an}_x\text{: separate } X) \text{ is} \]
do
\[ a := \textit{an}_x.a \]
end
...
\[ \textit{r (x)} \]
a.f

--- supplier
class X
feature
a: A
end
Validity rules – first attempt

- 4 SCOOP consistency 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!
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 (11)
- 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)
A few definitions...

Let $TypeId$ denote the set of declared type identifiers of a given Eiffel program. We define the set of tagged types for a given class as

$$TaggedType = OwnerId \times TypeId$$

where $OwnerId$ is a set of owner tags declared in the given class. Each class implicitly declares two owner tags: $\bullet$ (current processor) and $\perp$ (unknown).

The subtype relation $\prec$ on tagged types is the smallest reflexive, transitive relation satisfying the following axioms, where $\alpha$ is a tag, $S,T \in TypeId$, and $\prec_{Eiffel}$ denotes the subtype relation on $TypeId$:

$$(\alpha, T) \prec (\alpha, S) \iff T \prec_{Eiffel} S$$

$$(\alpha, T) \prec (\perp, T)$$
Specifying the type

class C

owner
   r1, r2   -- owner tags. • and ⊥ are declared implicitly.

feature
   a: A   -- a :: (•, A)
   x: separate X   -- x :: (⊥, X)
   y: separate Y within r1   -- y :: (r1, Y)

   r (an_x: separate X) is   -- an_x :: (⊥, X)
      do
         a := an_x.a
      end
   ...
end
And off we go!

\[
\begin{align*}
\text{[Current]} & \quad \Gamma \vdash \text{Current} :: (\bullet, T_{\text{Current}}) \\
\text{[DecNS]} & \quad l : T \in \Gamma \\
& \quad \Gamma \vdash l :: (\bullet, T) \\
\text{[DecSU]} & \quad l : \text{separate} T \in \Gamma \\
& \quad \Gamma \vdash l :: (\bot, T) \\
\text{[DecSO]} & \quad l : \text{separate} T \text{ within } r \in \Gamma \\
& \quad \Gamma \vdash l :: (r, T)
\end{align*}
\]
Assignment

\[\Gamma \vdash l :: (\alpha, T), \quad \Gamma \vdash e :: (\beta, S), \quad (\beta, S) \prec (\alpha, T)\]

\[\Gamma \vdash l := e\]

⇒

Separateness consistency rule (1)

If the source of an attachment (assignment instruction or argument passing) is separate, its target entity must be separate too.
Feature call

\[ \Gamma \vdash x :: (\alpha, T), \quad \Gamma \vdash a :: (\beta, S), \quad (\beta, S) \prec (\alpha_{fa}, T_{fa}), \]

\[ \text{[QCall]} \]

\[ \Gamma \vdash x.f(a) :: (\alpha_{fr}, T_{fr}) \]

*FormalArg* is the set of formal arguments of the routine where the expression is evaluated, \((\alpha_{fa}, T_{fa})\) is the type of the formal argument of feature \(f\) (we assume here that \(f\) has only one argument), \((\alpha_{fr}, T_{fr})\) is the type of its result.

We also define the type combinator

\[ * : \text{TaggedType} \times \text{TaggedType} \to \text{TaggedType} \]

\((\alpha, T) * (\beta, S) = \begin{cases} 
(\beta, S) & \text{if } \alpha = \bullet \\
(\alpha, S) & \text{if } \beta = \bullet \\
(\bot, S) & \text{otherwise}
\end{cases} \]
**Type combinator**

\[(\alpha, T) \star (\beta, S) = (\gamma, S)\]

<table>
<thead>
<tr>
<th></th>
<th>(\alpha)</th>
<th>(\perp)</th>
<th>(r_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\beta)</td>
<td>(\bullet)</td>
<td>(\perp)</td>
<td>(r_2)</td>
</tr>
<tr>
<td>(\perp)</td>
<td>(\perp)</td>
<td>(\perp)</td>
<td>(\perp)</td>
</tr>
<tr>
<td>(r_1)</td>
<td>(r_1)</td>
<td>(\perp)</td>
<td>(\perp)</td>
</tr>
</tbody>
</table>

- **non-separate calls preserve owner tag**
  - \(x :: (\alpha, T)\)
  - \(f :: (\beta, S)\)
  - \(x.f :: (\gamma, S)\)

- **separate calls always return separate type**
And now less formally.

\[
\begin{align*}
\Gamma \vdash x :: (\alpha, T), \quad \Gamma \vdash a :: (\beta, S), & \quad (\beta, S) \prec (\alpha, T) \ast (\alpha_{fa}, T_{fa}) \\
\alpha \neq \ast \Rightarrow x \in \text{FormalArg,} & \quad \alpha = \ast \Rightarrow \alpha_{fa} = \ast \\
\Gamma \vdash x.f(a) :: (\alpha, T) \ast (\alpha_{fr}, T_{fr}) & \quad \text{consistency rules 1 and 2} \\
\end{align*}
\]

\(x.f(a)\)

\text{FormalArg} is the set of formal arguments of the routine where the expression is evaluated, \((\alpha_{fa}, T_{fa})\) is the type of the formal argument of feature \(f\) (we assume here that \(f\) has only one argument), \((\alpha_{fr}, T_{fr})\) is the type of its result.

\[
(\alpha, T) \ast (\beta, S) = \begin{cases} 
(\beta, S) & \text{if } \alpha = \ast \\
(\alpha, S) & \text{if } \beta = \ast \\
(\bot, S) & \text{otherwise}
\end{cases}
\]
Consistency rules – second attempt

**Attachment rule (0)**
The type of the source of an attachment (assignment instruction or argument passing) must conform to the type of its target.

\[ x :: (\alpha, T), \quad y :: (\beta, S) \]
\[ x := y \quad \text{valid iff} \quad (\beta, S) \preceq (\alpha, T) \]

- But this attachment rule that already exists in the type system!

- The programmer just applies standard rule with augmented types.
Consistency rules – second attempt

**Expression type rule (1)**

Type of an expression (query call) $x.f$ depends on the type of its target ($x : : (\alpha, T)$) and the declared type of the query ($f : : (\beta, S)$).

$$x.f : : (\alpha, T)^*(\beta, S)$$

$$(\alpha, T)^*(\beta, S) = \begin{cases} 
(\beta, S) & \text{if } \alpha = \bullet \\
(\alpha, S) & \text{if } \beta = \bullet \\
(\bot, S) & \text{otherwise}
\end{cases}$$
Consistency rules – second attempt

Fully expanded types rule (2)
An entity $x$ that represents an object of a fully expanded type $FT$ (i.e. whose base class does not include, directly or indirectly, any non-separate attribute of reference) is seen as non-separate in any typing context.

$$x :: (\bullet, FT)$$

Basic types $INTEGER$, $BOOLEAN$, $CHARACTER$, $REAL$, etc. are fully expanded.
class \( X \)
owner
r1, r2
feature
a: X  -- a :: (\(\bullet\), X)
x: separate X  -- x :: (\(\perp\), X)
y: separate X within r1  -- y :: (r1, X)
z: separate Z within r1  -- z :: (r1, Z)

\[ x := a \]
-- valid because (\(\bullet\), X) is a subtype of (\(\perp\), X)
\[ a := x \]
-- invalid because (\(\perp\), X) is not a subtype of (\(\bullet\), X)
end
Examples: attachment rule

class X
owner
  r1, r2
feature
  a: X    -- a :: (•, X)
  x: separate X    -- x :: (⊥, X)
  y: separate X within r1    -- y :: (r1, X)
  z: separate Z within r1    -- z :: (r1, Z)

  x := y
  -- valid because (r1, X) is a subtype of (⊥, X)
  y := x
  -- invalid because (⊥, X) is not a subtype of (r1, X)
end
Examples: attachment rule

class X
owner
  r1, r2
feature
a: X    -- a :: (•, X)
x: separate X    -- x :: (⊥, X)
y: separate X within r1    -- y :: (r1, X)
z: separate Z within r1    -- z :: (r1, Z)

  y := a
  -- invalid because (•, X) is not a subtype of (r1, X)
  a := y
  -- invalid because (r1, X) is not a subtype of (•, X)

end
Examples: attachment rule

class X
owner
  r1, r2
feature
  a: X       -- a :: (·, X)
  x: separate X       -- x :: (⊥, X)
  y: separate X within r1       -- y :: (r1, X)
  z: separate Z within r1       -- z :: (r1, Z)
    -- assume that Z is a descendant of X

  y := z
-- valid because (r1, Z) is a subtype of (r1, X)
  z := y
-- invalid because (r1, X) is not a subtype of (r1, Z)
end
Examples: attachment rule

class X
owner
    r1, r2
feature
    a: A    -- a :: (•, A)
    x: separate X    -- x :: (⊥, X)
    y: separate X within r1    -- y :: (r1, X)
    z: separate Z within r1    -- z :: (r1, Z)

    r (an_x: separate X) is
        do ... end

    r (a)
    -- valid because (•, X) is a subtype of (⊥, X)
end
Examples: attachment rule

```plaintext
class X
  owner
    r1, r2
  feature
    a: A    -- a :: (•, A)
    x: separate X   -- x :: (⊥, X)
    y: separate X within r1    -- y :: (r1, X)
    z: separate Z within r1    -- z :: (r1, Z)

  r (an_x: separate X) is    -- an_x :: (⊥, X)
    do ... end

  r (x)
    -- valid because (⊥, X) is a subtype of (⊥, X)
end
```
Examples: attachment rule

```plaintext
class X
owner
    r1, r2
feature
a: A     -- a :: (∗, A)
x: separate X    -- x :: (⊥, X)
y: separate X within r1  -- y :: (r1, X)
z: separate Z within r1  -- z :: (r1, Z)

r (an_x: separate X) is     -- an_x :: (⊥, X)
do ... end

r (z)
-- valid because (r1, Z) is a subtype of (⊥, X)
end
```
class X
    owner
        r1, r2
    feature
        a: A -- a :: (•, A)
        x: separate X -- x :: (⊥, X)
        y: separate X within r1 -- y :: (r1, X)
        z: separate Z within r1 -- z :: (r1, X)

    s (an_x: X) is -- an_x :: (•, X)
        do ... end

    s (x)
        -- invalid because (⊥, X) is not a subtype of (•, X)
end
Examples: attachment rule

```plaintext
class X
owner
  r1, r2
feature
a: A    -- a :: (•, A)
x: separate X    -- x :: (⊥, X)
y: separate X within r1    -- y :: (r1, X)
z: separate Z within r1    -- z :: (r1, Z)

s (an_x: X) is -- an_x :: (•, X)
  do ... end

s (a)
-- valid because (•, X) is a subtype of (•, X)
end
```
Examples: expression type rule

class X
owner
  r1, r2
feature
a: A  -- a :: (\bullet, A)
x: separate X  -- x :: (⊥, X)
y: separate X within r1  -- y :: (r1, X)
r (an_x: separate X) is  -- an_x :: (⊥, X)
  do ... end

a := x.a
-- invalid because (⊥, A) is not a subtype of (\bullet, A)
x := y.x
-- valid because (⊥, X) is a subtype of (⊥, X)
end

Chair of Software Engineering
Examples: fully expanded types rule

```plaintext
class X
  owner
    r1, r2
  feature
    i: INTEGER    -- i :: (●, INTEGER)
    x: separate X  -- x :: (⊥, X)

  s (an_i: INTEGER) is    -- an_i :: (●, INTEGER)
    do ... end

  s (i)    -- obviously valid
  s (x.i)
    -- valid because x.i :: (●, INTEGER)
  x.s (i)
    -- valid
end
```

Piotr Nienaltowski, 24.05.2005
Why do we need explicit owners?

class X
feature
  y: Y
  set_y (a_y: Y) is
    do
      y := a_y
    end
end

-- in class C
r (x: separate X) is
local
  my_y: separate Y
  do
    my_y := x.y
    x.set_y (my_y) -- oops...
    -- set_y takes non-separate
    -- formal argument!
  end

---

Chair of Software Engineering
Piotr Nienaltowski, 24.05.2005
Specifying owners

- We need the possibility to say:

  *Entities* $x_1$, $x_2$, ..., $x_n$ *denote objects that are handled by the same processor.*

  We say that $x_1$, $x_2$, ..., $x_n$ *are owned by the same processor.*

```plaintext
class X
feature
  y: Y
  set_y (a_y: Y) is
    do
      y := a_y
    end
end

-- in class C
owner r1
feature
  r (x: separate X ∈ r1) is
    local
      my_y: separate Y ∈ r1
    do
      my_y := x.y
      x.set_y (my_y) -- it's alright now!
    end
end
```

Chair of Software Engineering  Piotr Nienaltowski, 24.05.2005
Another use of owner tags

class X
owner
  r1, r2
feature
x: separate X        -- x :: (⊥, X)
y, z: separate X within r1 -- y :: (r1, X), -- z :: (r1, X)

create x -- on which processor is x created?
  -- some fresh processor
create y
  -- y is created on (a fresh) processor denoted by r1
create z
  -- z is created on processor denoted by r1
end
meet_someone_e else's _friend (person: separate PERSON) is

local

a_friend: PERSON

do

a_friend := person.friend  -- Invalid assignment.
visit (a_friend)

end
function meet_someone_eles_friend (person: separate PERSON) is
local
  a_friend: PERSON

  do
    a_friend != person.friend -- Valid assignment attempt.
    if a_friend /= void then
      visit (a_friend)
    end
  end
Semantics of assignment attempt

instruction  \( l \neq e \)

with static types \( l :: (\alpha, T) \), \( e :: (\beta, S) \)
and dynamic type \( e :: (\beta_d, S_d) \)

is “equal” to:
\[
\text{if } (\beta_d, S_d) \prec (\alpha, T) \text{ then } l \leftarrow e \text{ else } l \leftarrow \text{void end}
\]

- Like in Eiffel but also downcasts owner tag
  - “deep downcast” over expanded attributes
Let \textit{TypeId} denote the set of declared type identifiers of a given Eiffel program. We define the set of tagged types for a given class as

\[
\text{TaggedType} = \text{OwnerId} \times \text{TypeId}
\]

where \textit{OwnerId} is a set of owner tags declared in the given class. Each class implicitly declares two owner tags: \bullet (\textit{current processor}) and \bot (\textit{undefined}).

The subtype relation \(<\) on tagged types is the smallest reflexive, transitive relation satisfying the following axioms, where \(\alpha\) is a tag, \(S, T \in \text{TypeId}\), and \(<_\text{Eiffel}\) denotes the subtype relation on \text{TypeId}:

\[
(\alpha, T) < (\alpha, S) \iff T <_\text{Eiffel} S
\]

\[
(\alpha, T) < (\bot, T)
\]
What we have learnt today

class $C$

owner
  r1, r2 -- owner tags. • and ⊥ are declared implicitly.

feature
a: A -- a :: (•, A)
x: separate X -- x :: (⊥, X)
y: separate Y within r1 -- y :: (r1, Y)

r (an_x: separate X) is -- an_x :: (⊥, X)
do
  a := an_x.a
end...
end
Attachment rule (0)

The type of the source of an attachment (assignment instruction or argument passing) must conform to the type of its target.

\[ x :: (\alpha, T), \quad y :: (\beta, S) \]
\[ x := y \quad \text{valid iff} \quad (\beta, S) \preceq (\alpha, T) \]
**Expression type rule (1)**

Type of an expression (query call) $x.f$ depends on the type of its target ($x :: (\alpha, T)$) and the declared type of the query ($f :: (\beta, S)$).

$$x.f :: (\alpha, T)^*(\beta, S)$$

$$(\alpha, T)^*(\beta, S) = \begin{cases} 
(\beta, S) & \text{if } \alpha = \bullet \\
(\alpha, S) & \text{if } \beta = \bullet \\
(\bot, S) & \text{otherwise}
\end{cases}$$

---

*Chair of Software Engineering*

Piotr Nienaltowski, 24.05.2005
Fully expanded types rule (2)

An entity $x$ that represents an object of a fully expanded type $FT$ (i.e. whose class does not declare any non-separate attribute of reference or not-fully-expanded type) is seen as non-separate in any typing context $r$. Basic types $INTEGER$, $BOOLEAN$, $CHARACTER$, $REAL$, etc. are fully expanded.
That’s all, folks!

Questions?