Concurrent Object-Oriented Programming

Bertrand Meyer, Volkan Arslan
Lecture 9: SCOOP: type system
Type system for SCOOP

- Prevents all traitors
  - static (compile-time) checks

- Simplifies, refines and formalises SCOOP rules

- Integrates expanded types and agents with SCOOP
  - More about it in future lecture

- Tool for reasoning about concurrent programs
  - May serve as basis for future extensions, e.g. for deadlock prevention schemes
Three components of a type

- Class type \( C \)
  \( x : X \)

- Processor tag \( \alpha \in \{ \bullet, T, \perp, <p>, <a.\text{handler}> \} \)

- Attached/detachable \( \gamma \in \{ !, ? \} \)

Current processor
\( x : \text{separate} \ X \)

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

No processor (bottom)
\( \text{Void} \)

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

\[ x : \text{X} \quad \text{--} \quad x :: (!, \bullet, \text{X}) \]

\[ y: \text{separate Y} \quad \text{--} \quad y :: (!, \tau, \text{Y}) \]

\[ z: ~ \text{? separate Z} \quad \text{--} \quad z :: (~, \tau, \text{Z}) \]

- Expanded types are attached and non-separate
  \[ i: \text{INTEGER} \quad \text{--} \quad i :: (!, \bullet, \text{INTEGER}) \]

- Void is detachable
  \[ \text{-- Void} :: (~, \perp, \text{NONE}) \]

- Current is attached and non-separate
  \[ \text{-- Current} :: (!, \bullet, \text{Current}) \]
Examples

\[x: \text{separate <px> X -- x :: (!, px, X)}\]

\[y: \text{separate <px> Y -- y :: (!, px, Y)}\]

\[z: \text{separate <px> Z -- z :: (!, px, Z)}\]

Entities \(x\), \(y\), \(z\) represent objects handled by the same processor known as \(px\).
Subtyping rules

- Since you do not like Greek letters, we will keep it informal

- $TT_2 \leq TT_1$ means “$TT_2$ is a subtype of $TT_1$”

- 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
  \[
  (\gamma, \bot, C) \leq (\gamma, \alpha, C)
  \]
So how does it help us?

- We can rely on standard type rules
- Enriched types give us additional guarantees
- Assignment rule: source conforms to target

\[
\Gamma |- x :: \text{TT}_x, \quad \Gamma |- e :: \text{TT}_e, \quad \Gamma |- \text{TT}_e \leq \text{TT}_x
\]

[Assign] \hspace{1cm} \Gamma |- x := e

- No need for special validity rules for separate
Examples (assignment)

\[
x: \ ?\text{separate} \ X \quad -- \text{where } Y \text{ conforms to } X
\]
\[
y: \ Y
\]
\[
z: \ \text{separate} \ X
\]
\[
my_z: \ X
\]

\[
x := y
\]
\[
x := z
\]
\[
y := x
\]
\[
z := x
\]
\[
my_z := z
\]
Examples (assignment)

\[
\begin{align*}
x &\colon \text{separate } X & \quad \text{-- } x &:: (?, T, X) \\
y &\colon Y & \quad \text{-- } y &:: (!, \bullet, Y) \\
z &\colon \text{separate } X & \quad \text{-- } z &:: (!, T, X) \\
my\_z &\colon X & \quad \text{-- } my\_z &:: (!, \bullet, X) \\
\end{align*}
\]

\[
\begin{align*}
x &:= y \\
x &:= z \\
y &:= x & \quad \text{-- invalid} \\
z &:= x & \quad \text{-- invalid} \\
my\_z &:= z & \quad \text{-- invalid} 
\end{align*}
\]
Examples

\[a: \text{separate } X\]
\[b: X\]
\[c: ? X\]
\[f(x, y: \text{separate } X) \text{ do } \ldots \text{ end}\]
\[g(x: X) \text{ do } \ldots \text{ end}\]
\[h(x: ? X): \text{separate } <p> X \text{ do } \ldots \text{ end}\]

\[f(a, b)\]
\[f(a, c)\]
\[g(a)\]
\[a := h(b)\]
\[a := h(a)\]
Examples

\[ a: \text{separate} \ X \quad -- \ a :: (!, \ T, \ X) \]
\[ b: X \quad -- \ b :: (!, \ ●, \ X) \]
\[ c: ? X \quad -- \ c :: (? , \ ●, \ X) \]

\[ f(x, y: \text{separate} \ X) \text{ do } \ldots \text{ end} \quad -- \ x :: (!, \ T, \ X) \quad -- \ y :: (!, \ T, \ X) \]
\[ g(x: X) \text{ do } \ldots \text{ end} \quad -- \ x :: (!, \ ●, \ X) \]

\[ h(x: ? X): \text{separate} \ <p> X \text{ do } \ldots \text{ end} \]
\[ -- x :: (? , \ ●, \ X) : (!, \ p, \ X) \]

\[ f(a, b) \]
\[ f(a, c) \quad -- \text{invalid} \]
\[ g(a) \quad -- \text{invalid} \]

\[ a := h(b) \]
\[ a := h(a) \quad -- \text{invalid} \]
Implicit types

An attached non-writable entity $e$ of type $T_e = (!, \alpha, C)$ also has an implicit type $T_{e, \text{imp}} = (!, e.\text{handler}, C)$

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

```
local

  z: \text{separate } Z

  do ... end
```

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

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

```
-- x :: (!, T, X) and an implicit type (!, x.\text{handler}, X)
-- s :: (!, ●, \text{STRING}) and an implicit type (!, s.\text{handler}, \text{STRING})
-- u :: (!, T, U) and an implicit type (!, u.\text{handler}, U)
-- y :: (?\text{, ●, } Y) and no implicit type because $y$ is detachable
-- z :: (!, T, Z) and no implicit type because $z$ is writable
```
Implicit types

In the context of class $C$, $Current$ has the type:

$Current :: (!, \bullet, C)$

$x :: X$

$x :: (!, Current.handler, X)$

hence $x :: \text{separate} <Current.handler> X$
Processor tags

- Processor tags are **always relative** to the current object.

- An entity declared as non-separate, e.g.\[a: A\]
is seen as **non-separate** by the current object.

- **Separate clients**, however, should see \(a\) as **separate**, because from their point of view it is **handled** by a different processor.

\[\Rightarrow \text{Type combinators necessary}\]
Type combinator * (result type)

Type $T_e$ of a query call $x.f(\ldots)$

$$T_e = T_{target} \ast T_{result}$$

where

$T_{target}$

$x$

$T_{result}$

result of $f$
Type combinator * (result type)

\[(\gamma, \alpha, C) * (\delta, \beta, D) = (\delta, \lambda, D)\]

<table>
<thead>
<tr>
<th></th>
<th>(\beta)</th>
<th></th>
<th>(\alpha)</th>
<th>(T)</th>
<th>(&lt;p&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\alpha)</td>
<td>(\cdot)</td>
<td>(T)</td>
<td>(T)</td>
<td>(T)</td>
<td>(T)</td>
</tr>
<tr>
<td>(\beta)</td>
<td>(\cdot)</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>(T)</td>
<td>(T)</td>
</tr>
</tbody>
</table>
Type combinator * (result type)

Result type:

\[ * : \text{Type} \times \text{Type} \rightarrow \text{Type} \]

\[ (\gamma, \alpha, C) * (\delta, \beta, D) = (\delta, \alpha, D) \text{ if } \beta \in \{\bullet, \text{Current.handler}\} \]

\[ = (\delta, T, D) \text{ otherwise} \]
An example

-- in class C

\(a: A\)

\(my\_x: X\)

\(r (x: \text{separate } X)\)

\hspace{1em} \text{do}

\hspace{2em} \text{x.f(a)} \text{ -- invalid, because traitor}

\hspace{1em} \text{end}

\hspace{1em} \ldots

\hspace{1em} \text{my\_x.f(a)} \text{ -- valid}

-- in class X

\(a: A\)

\(f (fa: ?A)\)

\hspace{1em} \text{do}

\hspace{2em} \ldots

\hspace{1em} \text{end}
Type combinator $\otimes$ (formals)

$f (fa: ?A) \ do \ ... \ end$

$x.f (a)$

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

where

$T_{target}$

$x$

$T_{formal}$

$fa$

$T_{actual}$

$a$
Type combinator $\otimes$ (formals)

$$(\gamma, \alpha, C) \otimes (\delta, \beta, D) = (\delta, \lambda, D)$$

<table>
<thead>
<tr>
<th></th>
<th>$\beta$</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha$</td>
<td>$\bullet$</td>
<td>$T$</td>
<td>$&lt;q&gt;$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\bullet$</td>
<td>$\bullet$</td>
<td>$T$</td>
<td>$\perp$</td>
</tr>
<tr>
<td>$T$</td>
<td>$\perp$</td>
<td>$T$</td>
<td>$\perp$</td>
<td></td>
</tr>
<tr>
<td>$&lt;p&gt;$</td>
<td>$&lt;p&gt;$</td>
<td>$T$</td>
<td>$\perp$</td>
<td></td>
</tr>
</tbody>
</table>
Type combinator $\otimes$ (formals)

Argument type:

$\otimes : \text{Type} \times \text{Type} \rightarrow \text{Type}$

$= (\delta, \alpha, D)$

if $\beta \in \{\bullet, \text{Current.handler}\} \land \alpha \neq T$

$(\gamma, \alpha, C) \ast (\delta, \beta, D) = (\delta, T, D)$

if $\beta = T$

$= (\delta, \bot, D)$

otherwise
Valid targets

-- in class $C$

$my_x, my_y$: separate $X$

$r (x: \text{separate } X)$

local

$y$: separate $X$

$do$

$x.f$ -- valid

$x.twin.f$ -- invalid although safe

$y := x.twin$

$y.f$ -- invalid although safe

$end$

-- continuation of class $C$

$s (x: \text{separate } X)$

$do$

$my_y.f$ -- invalid

$end$

... $r (my_x)$ $s (my_x)$
Call validity (informally)

- Informally, entity $x$ may be used as target of a feature call in the context of routine $r$
  - if and only if $x$ is attached and
  - processor that executes $r$ has exclusive access to $x$’s processor.
Unified rules for call validity

Definition (Valid target)

An expression $exp$ may be used as target of a feature call in the context of routine $r$ if and only if $exp$ is attached and satisfies at least one of the following conditions:

- $exp$ is non-separate
- $exp$ appears as formal argument of $r$
- $exp$ has a qualified processor tag $farg.handler$, and $farg$ is an attached formal argument of $r$
- $exp$ has an unqualified processor tag $p$, and some attached formal argument of $r$ has processor tag $p$. 
Unified rules for call validity

Definition (Controlled expression)

An expression \( \text{exp} \) of type \( T_{\text{exp}} = (\gamma, \alpha, C) \) is \textit{controlled} if and only if \( \text{exp} \) is \textit{attached} and satisfies \textit{one} of the following conditions:

- \( \text{exp} \) is non separate, i.e. \( \alpha = \bullet \)
- \( \text{exp} \) appears in a routine \( r \) that has an attached formal argument \( farg \) and \( \alpha = farg.\text{handler} \)
Feature call rules

Definition (Call validity rule)

Call \( \text{exp}.f(a) \) appearing class \( C \) is valid if and only if the following conditions hold:

- \( \text{exp} \) is controlled.
- \( \text{exp}'s\) base class has feature \( f \) exported to \( C \), and the actual arguments \( a \) conform in number and type to the formal arguments of \( f \).

- Type combinators necessary to calculate relative type
  - formal arguments \( \times \)
  - result \( \ast \)