Concurrent Object-Oriented Programming

Bertrand Meyer, Volkan Arslan
Lecture 13: SCOOP: Advanced techniques - Inheritance and Agents
Outline

SCOOP concurrency model

- Inheritance and polymorphism
- Agents
Inheritance and polymorphism

class A
feature
  x: separate X
  y: Y
end A

class B inherit A redefine x, y end
feature
  x: X
  y: separate Y
end B

-- in class C
r (x: separate X)
  require
    x.some_requirement
  do
    x.f
  ensure
    x.some_guarantee
end

s (y: Y)
  require
    y.some_requirement
  do
    y.f -- y is a traitor
  ensure
    y.some_guarantee
end

a: A
b: B
...
a := b -- Polymorphic assignment
r (a.x) -- Valid
s (a.y) -- Problematic
Inheritance and polymorphism

class A
feature
  r (x: separate X)
  feature
    require
      x.some_requirement
    do
      x.f
    ensure
      x.some_guarantee
  end

s (x: X)
  require
    x.some_requirement
  do
    x.f
  ensure
    x.some_guarantee
end
end

class B inherit A redefine r, s end
feature
  r (x: X)
  feature
    require
      x.some_requirement
    do
      x.f
      my_y := x
    end
end

  s (x: separate X)
  feature
    require
      x.some_requirement
    do
      x.f
    end
end

-- in class C
a: A
b: B
my_sep_x: separate X
my_x, my_y: X
a := b
-- Polymorphic assignment
a.r (my_sep_x)
-- Problematic
a.s (my_x)
-- Valid
Unified feature redefinition rule

The redefinition rules for the individual components of a type are now clarified:

- **Class types** may be redefined **covariantly** both in result types and argument types but the redefinition of a formal argument forces it to become **detachable**. For example, assuming that \( Y \) conforms to \( X \), an argument \( x: X \) may be redefined into \( x: ?Y \) but not \( x: Y \). An attribute may be redefined from \( my\_x: X \) into \( my\_x: Y \).

- **Detachable tags** may be redefined from \( ? \) to \( ! \) in result types. They may be changed from \( ! \) to \( ? \) in argument types, provided that no call on the redefined argument occurs in the original postcondition.

- **Processor tags** may be redefined from \( T \) to something more specific in result types, and from more specific to \( T \) in argument types.
Agents

- What are agents?

- Do we need special rules to handle them in SCOOP?
  - We hate special rules, don’t we?

- What agents can do for us
  - Convenience
  - Full asynchrony
What are agents?

- An agent represents an operation ready to be called

\[ x : X \]

\[ my\_operation : \text{ROUTINE} [X, \text{TUPLE}] \]

\[ my\_operation := \text{agent} \ x.f \]

\[ \ldots \]

\[ my\_operation.call ([]) \quad -- \text{Just like calling } x.f \]

- Agents can be created by one object, passed to another one, and called by the latter

\[ y.r (my\_operation) \quad -- y \text{ will call agent later on} \]
What are agents?

- Arguments can be closed (fixed) or open

```plaintext
my_operation := agent io.put_string ("Hello World!")
my_operation.call ([])  -- no arguments necessary;
                          -- use empty tuple

my_operation := agent io.put_string (?)
my_operation.call (['"Hello World!"'])  -- 1 argument
```

- Based on generic classes

```
ROUTINE [BASE_TYPE, OPENARGS -> TUPLE]

PROCEDURE [BASE_TYPE, OPENARGS -> TUPLE]

FUNCTION [BASE_TYPE, OPENARGS -> TUPLE, RESULT_TYPE]
```
Use of agents

- Object-oriented wrappers for operations
  - > strongly-typed function pointers (C++)
  - ~ .NET delegates

- Used in event-driven programming
  - Subscribe an action to an event type
  - The action is executed when event occurs

- Loose coupling of software components
  - Model - View - Controller

- Replace several patterns
  - Observer
  - Visitor
  - ...
Problematic agents

my_agent: PROCEDURE [separate ANY, TUPLE]
   x: separate X

   my_agent := agent x.f
   my_agent.call ([[]]) -- Like x.f without locking x!

\[
\begin{align*}
\text{my\_agent: } & \text{PROCEDURE [separate ANY, TUPLE]} \\
x: & \text{separate } X \\
\vdots & \\
\text{my\_agent := agent x.f} \\
\text{my\_agent.call ([[]])} & \text{-- Like x.f without locking x!}
\end{align*}
\]
Let’s make the agent separate!

my_agent: separate PROCEDURE [X, TUPLE]

x: separate X

\ldots

my_agent := agent x.f

-- agent x.f handled by x’s processor

my_agent.call ([]) -- Invalid call!
Separate agents

- Agent built on a separate call becomes itself separate
  - It is handled by the same processor as its target

\[
\text{my\_agent: separate PROCEDURE } [X, \text{TUPLE}] \\
x: \text{separate } X \\
\ldots \\
\text{my\_agent := agent } x.f \\
\text{call (my\_agent)} \\
\text{call (an\_agent: separate PROCEDURE } [\text{ANY, TUPLE}]) \\
\quad \text{do} \\
\quad \quad \text{an\_agent.call ([]} \\n\quad \quad \quad \quad \text{-- Valid separate call} \\
\quad \text{end}
\]

- No special rules for separate agents
- Agents pass processors’ boundaries just as other objects do
1st benefit: convenience

- Without agents, enclosing routines are necessary for every separate call
  
  ```
  my_x: separate X
  r (my_x) -- x.f
  s (my_x) -- x.g (5, "Hello")
  ```
  
  ...  
  ```
  r (x: separate X)
  do
  x.f
  end
  ```  
  ```
  s (x: separate X)
  do
  x.g (5, "Hello")
  end
  ```

- With agents, we can write universal enclosing routine
  ```
  call (agent my_x.f)
  call (agent my_x.g (5, "Hello"))
  ```
  ```
  call (an_agent: separate PROCEDURE [ANY, TUPLE])
  -- Universal enclosing routine.
  do
  an_agent.call ([])  
  en
  ```
2nd benefit: full asynchrony

- Without agents, full asynchrony cannot be achieved
  
  \[
  \text{my}_x, \text{my}_y : \text{separate } X
  \]

  ... 
  \[
  r (\text{my}_x) \quad \text{-- Blocking call}
  \]
  \[
  \text{do}_\text{local}_\text{stuff}
  \]

  ... 
  \[
  r (x : \text{separate } X)
  \]
  \[
  \quad \text{do}
  \]
  \[
  \quad \quad \text{x.f} \quad \text{-- Asynchronous}
  \]
  \[
  \quad \text{end}
  \]

- With agents, it’s easy-peasy
  
  \[
  \text{asynch (agent } \text{my}_x.f) \quad \text{-- Non-blocking call}
  \]
  \[
  \text{do}_\text{local}_\text{stuff}
  \]

  ... 
  \[
  \text{asynch (an_agent: } ?\text{separate PROCEDURE [ANY, TUPLE]})
  \]
  \[
  \quad \text{-- Call an_agent asynchronously.}
  \]
  \[
  \quad \text{do}
  \]
  \[
  \quad \quad \text{...}
  \]
  \[
  \quad \text{end}
  \]
How to achieve full asynchrony

\[ \text{asynch (agent my\_x.f)} \]

\[ \text{do\_local\_stuff} \]

\[ \text{P}_1 \]

\[ \text{my\_x} \]

\[ \text{(CLIENT)} \]

\[ \text{P}_2 \]

\[ \text{target} \]

\[ \text{(PROCEDURE [X, TUPLE])} \]

\[ \text{P}_3 \]

\[ \text{feature\_to\_execute} \]

\[ \text{(EXECUTOR)} \]

\text{separate reference}

\text{non-separate reference}
Separate executors

- Feature \textit{asynch} implemented in class \textit{CONCURRENCY}

\texttt{asynch (agent my\_x\_f)}

\texttt{asynch (an\_agent: \textbf{?separate} PROCEDURE [ANY, TUPLE])}

- Call `an_agent' asynchronously.
- Note that `an_agent' is not locked.

\texttt{local}

\texttt{executor: separate EXECUTOR}

\texttt{do}

\texttt{create executor.make (an\_agent)}

\texttt{launch (executor)}

\texttt{end}

- Asynchronous calls on non-separate targets (including \textit{Current})

\texttt{asynch (agent f)}

- Call \textit{Current.f} asynchronously.
- It will be executed when current processor becomes idle.