Concurrent Object-Oriented Programming

Bertrand Meyer, Piotr Nienaltowski
Lecture 12-13:
Advanced OO techniques in SCOOP
Outline

- Once functions
- Inheritance
  - Polymorphism
  - Deferred classes
  - Feature redefinition
  - Precursor calls
- Genericity
  - Constrained genericity
- Agents
Once functions

\textit{barrier: separate \texttt{MY\_BARRIER}}

\begin{verbatim}
  -- Barrier.
  once
  create Result.make (3)
end
\end{verbatim}

- Similar to constants
  - Always return the same value

- Lazy evaluation
  - Body executed on first access
  - Once-per-class semantics

- Examples of use
  - Heavy computations
    - Stock market statistics
  - Common contact point for objects of one type
    - Feature \texttt{io} in class \texttt{ANY}
    - Queue for meeting Santa
Once functions in concurrent context

- Is once-per-system semantics always correct?

\[
\begin{align*}
\text{barrier: separate } & \text{MY\_BARRIER} \\
& \text{-- Barrier.} \\
& \text{once} \\
& \text{create Result.make (3)} \\
& \text{end}
\end{align*}
\]

\[
\begin{align*}
\text{local\_printer: PRINTER} \\
& \text{once} \\
& \text{printer\_pool.item (Current.location)} \\
& \text{end}
\end{align*}
\]

- Separate functions are once-per-system
- Non-separate functions are once-per-processor
Scoop2scooooli and once functions

- Preprocessor applies the appropriate semantics

```plaintext
barrier: separate MY_BARRIER is
  once
  . . .
end

becomes

barrier: SCOOP_SEPARATE__MY_BARRIER is
  indexing
  once_status: global
  once
  . . .
end
```
Inheritance

- Polymorphism
- Deferred classes
- Feature redefinition
- Precursor calls
Polymorphism

class ELF inherit SANTA_HELPER ... end

class REINDEER inherit SANTA_HELPER ... end

class SANTA

... feature
    elf: separate ELF
    reindeer: separate REINDEER

... meet_helper (a_helper: separate SANTA_HELPER) is
    require
        a_helper.is_ready
    do
        a_helper.talk_it_over (Current)
    end

... meet_helper (elf)                -- Polymorphic attachment
    meet_helper (reindeer)            -- Polymorphic attachment

... end
class SANTA
  ...
  feature
    elf: separate ELF
    local_elf: ELF
    ...
    meet_helper (a_helper: separate SANTA_HELPER) is
      require
        a_helper.is_ready
      do
        a_helper.talk_it_over (Current)
      end
    ...
    meet_helper (elf)  -- This works.
    meet_helper (local_elf)  -- This doesn’t because it requires
                              -- both polymorphism and conversion.
    ...
  end
Solution

- First solution: conversion followed by polymorphism

  ```
  aux_elf: separate ELF
  local_elf: ELF
  ...
  aux_elf := local_elf       -- Conversion.
  meet_helper (aux_elf)     -- Polymorphism.
  ```

- Second solution: polymorphism followed by conversion

  ```
  aux_santa_helper: SANTA_HELPER
  local_elf: ELF
  ...
  aux_santa_helper := local_elf    -- Polymorphism.
  meet_helper (aux_santa_helper)  -- Conversion.
  ```

- Second solution only works if `SANTA_HELPER` is not deferred
Deferred classes

**deferred class** SANTA HELPER . . . End

- Conversion cannot be used on deferred types
  ```
  local_santa_helper: SANTA HELPER
  santa_helper: separate SANTA HELPER
  . . .
  santa_helper := local_santa_helper   -- Invalid.
  ```

- Solution: use non-deferred subtype of SANTA HELPER
Feature redefinition

- Usual rules of DbC apply
- Preconditions may be kept or weakened
- Postconditions and invariants may me kept or strengthened
- Result types may be redefined covariantly
- Formal argument types may be redefined
  - Covariantly w.r.t. class type
  - Contravariantly w.r.t. attached tag and processor tag
Inheritance: preconditions

- Preconditions may be kept or weakened
  - Less waiting
  - Full support for preconditions in scoop2scoopl

```
-- ancestor
r (x: separate X )
  require
    x.count > 10
  do
    ...
  end
r (my_x)

-- descendant
r (x: separate X )
  require else
    x.count > 0
  do
    ...
  end
```

- Contracts are bound dynamically
  - demo
Inheritance: postconditions and invariants

- Postconditions may be kept or strengthened
  - More guarantees to the client
  - Scoop2scoopli applies sequential semantics
  - Postcondition checking turned off by default

- Invariants may be kept or strengthened
  - Full support for invariant checking in scoop2scoopli
  - Invariant checking turned on by default

- See “Contracts for concurrency” paper on the website
Inheritance: postconditions and invariants

- Result types may be redefined covariantly

  -- ancestor
  \textit{my\_x: separate} \ X
  \begin{verbatim}
  r (x: separate X)
  do
    ... 
  end
  \end{verbatim}

  -- descendant
  \textit{my\_x: X}
  \begin{verbatim}
  r (x: separate X)
  do
    ... 
  end
  \end{verbatim}

  \textit{r (my\_x)}

  -- Blocking call

  -- Ancestor’s \textit{r (my\_x)} is now a non-blocking call.

  -- Is it a problem?

- No problem here: client waits less
Argument types: attached tags

- Formal argument types may be redefined
  - Covariantly w.r.t. class type
  - Contravariantly w.r.t. attached tag and processor tag

  -- ancestor
  my_y: Y

  -- descendant
  my_y: Y

  r (y: separate Y)
  do
  . . .
  end

  r (y: ? separate Y)
  do
  . . .
  end

  r (my_y)
  -- Blocking call

  -- Ancestor’s r (my_y) is now a
  -- non-blocking call.

  -- Is it a problem?

- No problem here: client waits less
Argument types: processor tags

- Formal argument types may be redefined
  - Covariantly w.r.t. class type
  - Contravariantly w.r.t. attached tag and processor tag

```
-- ancestor
my_y: Y

r (y: Y)
do
  ...
end

-- descendant
my_y: Y

r (y: separate Y)
do
  ...
end
```

- No problem here: client may only use non-separate actual
  - Why?
  - Is r (my_y) really blocking?
Precursor calls

- Precursor call invokes ancestor's version of redefined routine.
- Allows reuse of routine implementation.

```plaintext
class ELF
inherit SANTA_HELPER redefine make end
feature -- Initialization
  make (an_id: INTEGER)
    require
      an_id > 0
    do
      Precursor {SANTA_HELPER} (an_id)
      -- Additional stuff here.
    end
  ...
end
```
Scoop2scooooli and precursor calls

- Scoop2scooooli uses renaming to implement dynamic binding of routine bodies and wait-conditions

- Therefore, explicit ancestor annotations are required

**Precursor** \{SANTA HELPER\} (an\_id) -- Valid.

**Precursor** (an\_id) -- Invalid.

- This only applies to routines that take separate arguments

- It is a good habit to indicate the ancestor anyway.
Scoop2scoopli: feature redefinition

- Preconditions: fully supported
- Postconditions: supported but sequential semantics applies
- Invariants: fully supported
- Redefinition of class types: fully supported
- Redefinition of processor tags and detachable tags: not supported.
- Precursor calls: supported but explicit ancestor name required
- See “Flexible locking in SCOOP” paper on the website
Genericity

- Entities of generic types may be separate

  \[\text{list: LIST [X]}\]

  \[\text{list: separate LIST [X]}\]

- Actual generic parameters may be separate

  \[\text{list: LIST [separate X]}\]

  \[\text{list: separate LIST [separate X]}\]

- All combinations are meaningful and useful
Genericity: even more fun

- Actual generic parameters may be also of generic type

  \[
  \text{list: } \text{LIST [} \text{separate ARRAY [} \text{SET [} \text{separate } X] \text{]}\text{]}
  \]

- Separateness is relative to object of generic class, e.g. elements of

  \[
  \text{list: } \text{separate LIST [} X \text{]}
  \]

are non-separate w.r.t. \text{list} but separate w.r.t. \textbf{Current}.

- Type combinators apply
Scoop2scoopli and genericity

- Full support
  - list: LIST [X]
  - list: LIST [separate X]
  - list: separate LIST [X]  -- X expanded
  - list: separate LIST [separate X]

- No support
  - list: separate LIST [X]  -- X non-expanded

But it can be simulated with
  - list: separate LIST [separate X]
class PRIORITY_QUEUE [G -> PART_COMPARABLE]

- Actual generic parameter must conform
  queue: separate PRIORITY_QUEUE [X]
  -- X must inherit from PART_COMPARABLE

- Scoop2scoopli allows
  queue: PRIORITY_QUEUE [X]
  queue: separate PRIORITY_QUEUE [X]

  but disallows
  queue: PRIORITY_QUEUE [separate X]
  queue: separate PRIORITY_QUEUE [separate X]
Constrained genericity

- Try
  class MY_QUEUE [G -> separate SANTA_HELPER]

  q: separate MY_QUEUE [separate REINDEER]

- Scoop2scoopli accepts it

- Precompiled code compiles correctly

- ES syntax checker complains
  - Just ignore it
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
  - Parallel wait

- Demo
What are agents?

- An agent represents an operation ready to be called
  \[ x: X \]
  \[ \text{my\_operation: ROUTINE } [X, \text{TUPLE}] \]

  \[ \text{my\_operation := agent x.f} \]
  \[ \ldots \]
  \[ \text{my\_operation.call ([])} \] -- Just like calling x.f

- Agents can be created by one object, passed to another one, and called by the latter
  \[ y.r(my\_operation) \] -- y will call agent later on
What are agents?

- Arguments can be **closed** (fixed) or **open**

  ```
  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, OPEN_ARGS -> TUPLE]
  PROCEDURE [BASE_TYPE, OPEN_ARGS -> TUPLE]
  FUNCTION [BASE_TYPE, OPEN_ARGS -> 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
  - ...
Problemsatic agents

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

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

Chair of Software Engineering
Let’s make the agent separate!

`my_agent: separate PROCEDURE [X, TUPLE]`

`x: separate X`

```
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`

```plaintext
my_agent: separate PROCEDURE [X, TUPLE]
x: separate X
...
my_agent := agent x.f

call (my_agent)

call (an_agent: separate PROCEDURE [ANY, TUPLE])
do
  an_agent.call ([])  -- Valid separate call
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

\[
\begin{align*}
\text{my\_x: separate } & X \\
r (\text{my\_x}) & -- x.f \\
s (\text{my\_x}) & -- x.g (5, "Hello") \\
\end{align*}
\]

...  

\[
\begin{align*}
r (x: \text{separate } X) & \\
\quad \text{do} & \\
\quad \quad x.f & \\
\quad \text{end} & \\
s (x: \text{separate } X) & \\
\quad \text{do} & \\
\quad \quad x.g (5, "Hello") & \\
\quad \text{end} &
\end{align*}
\]

- With agents, we can write universal enclosing routine

\[
\begin{align*}
\text{call (agent my\_x.f)} & \\
\text{call (agent my\_x.g (5, "Hello"))} &
\end{align*}
\]

\[
\begin{align*}
call (\text{an\_agent: separate \text{PROCEDURE \{\text{ANY, TUPLE}\}}}) & \\
\quad -- \text{Universal enclosing routine.} & \\
\quad \text{do} & \\
\quad \quad \text{an\_agent.call ([])} & \\
\quad \text{end} &
\end{align*}
\]
2\textsuperscript{nd} benefit: full asynchrony

- **Without agents, full asynchrony cannot be achieved**
  
  \texttt{my\_x, my\_y: separate X}
  
  \ldots
  
  \texttt{r (my\_x)} -- Blocking call
  
  \texttt{do\_local\_stuff}
  
  \ldots
  
  \texttt{r (x: separate X)}
    
    \texttt{do}
      
      \texttt{x.f} -- Asynchronous
    
    \texttt{end}

- **With agents, it's easy-peasy**
  
  \texttt{async (agent my\_x.f)} -- Non-blocking call
  
  \texttt{do\_local\_stuff}
  
  \ldots
  
  \texttt{async (an\_agent: ?separate PROCEDURE [ANY, TUPLE])}
    
    -- Call `an\_agent` asynchronously.
    
    \texttt{do}
      
      \ldots
    
    \texttt{end}
How to achieve full asynchrony

\[
\text{asynch (agent my\_x.f)} \quad \text{do\_local\_stuff}
\]
Separate executors

- Feature `async` implemented in class `CONCURRENCY`

```plaintext
async (agent my_x.f)

async (an_agent: ?separate PROCEDURE [ANY, TUPLE])
  -- Call `an_agent` asynchronously.
  -- Note that `an_agent` is not locked.

local
  executor: separate EXECUTOR

do
  create executor.make (an_agent)
  launch (executor)
end
```

- Asynchronous calls on non-separate targets (including `Current`)

```plaintext
async (agent f)
  -- Call `Current.f` asynchronously.
  -- It will be executed when current processor becomes idle.
```
3rd benefit: waiting faster

\[ my_x, my_y: \text{separate } X \]
\[ \ldots \]
\[ \text{if } my_x.b \text{ or else } my_y.b \text{ then } \ldots \text{ end} \]
\[ \text{if or else } (my_x, my_y) \text{ then } \ldots \text{ end} \]

\text{or else } (x, y: \text{separate } X): \text{BOOLEAN}
\[ \text{do} \]
\[ \text{Result := } x.b \text{ or else } y.b \]
\[ \text{end} \]

- What if \( my_x \) or \( my_y \) is busy?
- What if \( my_x.b \) is false but \( my_y.b \) is true?
- What if evaluation of \( my_x.b \) takes ages whereas \( my_y.b \) evaluates very fast?
my_x.b or else my_y.b
if parallel_or (agent my_x.b, agent my_y.b) then ... End

parallel_or (a1, a2: ?separate FUNCTION [ANY, TUPLE, BOOLEAN]): BOOLEAN
-- Result of `a1' or else `a2' computed in parallel.
local
  answer_collector: separate ANSWER_COLLECTOR [BOOLEAN]
do
  create answer_collector.make (a1, a2)
  Result := answer (answer_collector)
end
answer (a_collector: separate ANSWER_COLLECTOR [ANY]): ANY
-- Result returned by `an_answer_collector'.
require
  answer_ready: a_collector.is_ready
do
  Result ?= a_collector.answer
end
Generalised mechanism

- Parallel or, parallel and, ...
- Launch n jobs and wait for first result, etc.
- Implemented in class `CONCURRENCY`
- Relies on generic classes
  - `ANSWER_COLLECTOR [RESULT_TYPE]`
  - `EVALUATOR [RESULT_TYPE]`
- For the moment it’s a pattern; I’ll turn it into a component
make
  -- Creation procedure.

local
  x, y: separate X
  a, b: separate ORACLE

do
  -- testing asynchrony
  create x; create y
  async (agent x.print_info)
  async (agent Current.print_info)
  async (agent y.print_info)

  -- testing parallel wait
  create a.make (False, 10000); create b (True, 0)
  res := parallel_or (agent a.boolean_query,
                        agent b.boolean_query)

end
Conclusions

- **Agents and concurrency**
  - Tricky at first; easy in the end
  - Agents built on separate calls are separate
  - Open-target agents are non-separate on creation
  - Agents treated just like any other object

- **Advantages brought by agents**
  - *Convenience:* “universal” enclosing routine for single calls
  - Full asynchrony: non-blocking calls
  - Truly parallel wait
  - All these are implemented as library mechanisms

- **Support in scoop2scoopl**
  - Limited
  - You’d need to hack preprocessed code
That’s all, folks!

- **SCOOP**
  - Computational model, synchronisation.
  - Traitors, validity rules.
  - Type system for SCOOP, attached and detachable types, lock passing.
  - Advanced O-O techniques: inheritance, polymorphism, genericity, agents.
  - Santa, Santa, Santa, ...

- **Traditional approaches**
  - Safety, liveness, fairness.
  - Atomicity violations, deadlock, livelock.
  - Mutexes, semaphores, barriers, monitors, CCRs, rendezvous.
  - Protected objects, Actors.
  - Active objects, inheritance anomalies.
Questions?

- Did I hear “exam”? I knew you’d ask that question. Here’s the answer:

- **SCOOP (easy questions)**
  - Computational model, synchronisation.
  - Traitors, validity rules.
  - Type system for SCOOP, attached and detachable types, lock passing.
  - Advanced O-O techniques: inheritance, polymorphism, genericity, agents.
  - Santa, Santa, Santa, ...

- Traditional approaches (more difficult questions)
  - Safety, liveness, fairness.
  - Atomicity violations, deadlock, livelock.
  - Mutexes, semaphores, barriers, monitors, CCRs, rendezvous.
  - Protected objects, Actors.
  - Active objects, inheritance anomalies.