Concurrent Object-Oriented Programming

Prof. Dr. Bertrand Meyer

Lecture 10: Advanced Object-Oriented Mechanisms
(based on work with Piotr Nienaltowski)
What is an agent?

- An agent represents an operation ready to be called.
  
  \[
  x : X \\
  \text{op1: ROUTINE [X, TUPLE]} \\
  \text{op1 := agent } x . f \\
  \text{op1.call ([])}
  \]

- Agents can be created by one object, passed to another one, and called by the latter
What is an agent?

• Arguments can be closed (fixed) or open.
  
  ```
  op1 := agent io.put_string ("Hello World!")
  op1.call ([[]])
  ```

  Empty tuple as argument

  ```
  op1 := agent io.put_string (?)
  op1.call (["Hello World!"])  
  ```

  One-argument tuple

• They are 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++)
- Similar to .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
Replace several patterns
- Observer
- Visitor
- Model - View - Controller

...
Problematic Agents

- Which processor should handle an agent? Is it the target processor or the client processor?
- Let's assume it is the client processor.

\[ a1: \text{PROCEDURE} [\text{separate ANY, TUPLE}] \]
\[ x: \text{separate } X \]
\[ \ldots \]
\[ a1 := \text{agent } x.f \]
\[ a1.\text{call } ([]) \]

Like \( x.f \) without locking \( x \)
Let’s make the agent separate!

- The agent needs to be on the target processor.

```
a1: separate PROCEDURE [X, TUPLE]
x: separate X
...
a1 := agent x.f
a1.call [[])
```

This agent will be handled by x’s processor

Invalid
Let’s make the agent separate!

• No special type rules for separate agents
• Semantic rule: an agent is created on its target’s processor
• Agents pass processors’ boundaries just as other objects do

```plaintext
a1: separate PROCEDURE [X, TUPLE]
x: separate X
a1 := agent x.f

call (a1)
call (an_agent: separate PROCEDURE [ANY, TUPLE])
  do
    an_agent.call ([])
  end
```

Valid separate call
First benefit: convenience

- Without agents, enclosing routines are necessary for every separate call.

\[
x1: \text{separate } X \\
\text{r (x: separate X) s (x: separate X)} \\
\text{do} \\
\text{do} \\
\text{x.f} \\
\text{x.g (5, “Hello“)} \\
\text{end} \\
\text{end}
\]

- With agents, we can write a universal enclosing routine.

\[
\text{call (agent x.f); call (agent x.g (5, “Hello“))}
\]

\[
\text{call (an_agent: separate \text{PROCEDURE [ANY, TUPLE]})}
\]

\[
\quad \text{-- Universal enclosing routine.}
\]

\[
\quad \text{do}
\]

\[
\quad \text{an_agent.call ([])}
\]

\[
\quad \text{end}
\]
Second benefit: full asynchrony

- **Without agents, full asynchrony cannot be achieved**
  
  ```
  x1, y1: separate X  
  r (x: separate X)  
  do  
  x.f  
  end  
  ```

- **With agents it works**
  
  ```
  async (agent x1.f)  
  do_local_stuff  
  ```

  ```
  async (a: detachable separate PROCEDURE [ANY, TUPLE])  
  -- Call a asynchronously.  
  do  
  ...  
  end  
  ```
Full asynchrony

- An executor object gets created on another processor to execute the agent asynchronously.

![Diagram showing full asynchrony](image)
Full asynchrony

• The feature `asynch` is implemented in the class `CONCURRENCY`.

```plaintext
asynch (an_agent: detachable 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
```

• An asynchronous call on a non-separate targets (including `Current`) will be executed when the current processor becomes idle.
Separate agent benefit 3: waiting faster

\[
x_1, y_1: \text{separate } X \\
\text{or}_\text{else} (x, y: \text{separate } X): \text{BOOLEAN} \\
\quad \text{do} \\
\quad \text{if } \text{or}_\text{else} (x_1, y_1) \text{ then} \\
\quad \quad \ldots \\
\quad \text{end} \\
\quad \text{Result := } x.b \text{ or else } y.b \\
\text{end}
\]

- What if \( x_1 \) or \( y_1 \) is busy?
- What if \( x_1.b \) is false but \( y_1.b \) is true?
- What if evaluation of \( x_1.b \) takes ages whereas \( y_1.b \) evaluates very fast?
Waiting faster

if parallel_or (agent x1.b, agent y1.b) then

...
end

parallel_or (a1, a2: detachable 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 (ac: separate ANSWER_COLLECTOR [BOOLEAN]): BOOLEAN

-- Result returned by `an_answer_collector'.

require

answer_ready: ac.is_ready
do
Result := ac.answer
end
Waiting faster

P1 P2 P3

x1 y1 ans_col
(CLIENT)

b (X)
target
(FUNCTION [...])
target
(FUNCTION [...])

query
answer_c_collector
(EVALUATOR [...])

answer
(ANSWER_COLLECTOR [...])

P4 P5 P6
3rd Benefit: Waiting Faster

- 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]
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
Once Functions

• Similar to constants
  • Always return the same value
• Lazy evaluation
  • Body executed on first access
• Once per thread or once per object semantic.
• Examples of use
  • Heavy computations
    • Stock market statistics
  • Common contact point for objects of one type
    • Feature io in class ANY
Once functions in a concurrent context

• Is once-per-system semantics always correct?

  barrier: separate BARRIER
  once
  create Result.make (3)
  end

  local_printer: PRINTER
  once
  printer_pool.item (Current.location)
  end

• Separate functions are once-per-system.
• Non-separate functions are once-per-processor.
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.