Concepts of Concurrent Computation

Bertrand Meyer
Sebastian Nanz

Lecture 9: SCOOP Advanced Mechanisms
& Other Concurrency Models
What is an agent?

- An agent represents an operation ready to be called.
  ```
  x: X
  op1: ROUTINE [X, TUPLE]
  op1 := agent x.f
  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.

\[
\begin{align*}
\text{a1: PROCEDURE [separate ANY, TUPLE]} \\
\text{x: separate X} \\
\text{...} \\
\text{a1 := agent x.f} \\
\text{a1.call ([])}
\end{align*}
\]
Let’s make the agent separate!

- The agent needs to be on the target processor.

\[a1 := \text{agent } x.f\]

\[x: \text{separate } X\]

\[\ldots\]

\[\text{This agent will be handled by } x\text{'s processor}\]

\[a1: \text{separate } \text{PROCEDURE} [X, \text{TUPLE}]\]
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.

\[
\begin{align*}
\text{x1: separate } X & \quad \text{r (x: separate } X) \quad \text{s (x: separate } X) \\
\text{r (x1)} & \quad \text{do} \\
\text{s (x1)} & \quad \text{do} \\
\text{x.f} & \quad \text{x.g (5, "Hello")} \\
\text{end} & \quad \text{end}
\end{align*}
\]

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

\[
\begin{align*}
\text{call (agent x1.f); call (agent x1.g (5, "Hello"))}
\end{align*}
\]

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

- Without agents, full asynchrony cannot be achieved
  
  ```
  x1, y1: separate X
  r (x: separate X)
  r (x1)
  do_local_stuff
  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.
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.
Third benefit: waiting faster

\[ x_1, y_1: \text{separate } X \]

\[ \text{or\_else } (x, y: \text{separate } X): \text{BOOLEAN} \]

\[ \text{do} \]

\[ \text{if or\_else } (x_1, y_1) \text{ then} \]

\[ \ldots \]

\[ \text{end} \]

\[ \text{Result} := x.b \text{ or else } y.b \]

\[ \text{end} \]

- 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 (1)

```plaintext
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 (2)
Waiting Faster (3)

- 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]`
Agents wrap-up

- **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?

```plaintext
barrier: separate BARRIER                              local_printer: PRINTER
         once                                              once
   create Result.make (3)                               printer_pool.item (Current.location)
end                                                 end
```

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

- Entities of generic types may be separate
  
  \text{list}: \text{LIST} [\text{BOOK}]
  
  \text{list}: \text{separate LIST} [\text{BOOK}]

- Actual generic parameters may be separate
  
  \text{list}: \text{LIST} [\text{separate BOOK}]
  
  \text{list}: \text{separate LIST} [\text{separate BOOK}]

- All combinations are meaningful and useful
- Separateness is relative to object of generic class, e.g. elements of \text{list}: \text{separate LIST} [\text{BOOK}] are non-separate with respect to (w.r.t) \text{list} but separate w.r.t. \textbf{Current}. Type combiners apply.
More concurrency models

Polyphonic C#

(Slides thanks to C.A. Furia)
Introducing Polyphonic C#

- **Polyphonic C#** is an extension of C# with a few high-level primitives for concurrency
  - Based on join calculus (Fournet & Gonthier, 1996)
  - Taken up by Microsoft’s Cω project
  - **JoinJava** is a similar extension for Java
- Based on two basic notions
  - Asynchronous methods
  - Chords

*(M. Mussorgsky, Pictures at an exhibition)*
Asynchronous methods

Calls to asynchronous methods return immediately without returning any result

- The callee is scheduled for execution in a different thread
- similar to sending a message or raising an event
- declared using `async` keyword instead of `void`

```csharp
public async startComputation () {
    // computation
}
```

- asynchronous methods do not return any value
Chords: syntax

A chord is an extension of the notion of method definition

• The signature of a chord is a collection of (traditional) method declarations joined by &
• The body of a chord is all similar to the body of a traditional method

```java
public int get() & public async put(int i) {
    return i;
}
```

• Within a chord:
  • at most one method can be non-asynchronous
• Within a class:
  • the same method can appear in more than one chord
Chords: semantics

A chord is only executed once all the methods in its signature have been called

- Calls are buffered until there is a matching chord
  - the implicit buffer supports complex synchronization patterns with little code (see Producer/Consumer later)
- If multiple matches are possible, nondeterminism applies
- Execution returns a value to the only non-asynchronous method in the chord (if any)
Chords semantics: example

```java
public class Buffer() {
    public int get() & public async put(int i) {
        return i;
    }
}

Buffer b = new Buffer();
b.put("okey")
Console.WriteLine(b.get()); // prints "okey"
b.put("okey"); b.put("dokey");
    // prints "okeydokey" or "dokeyokey"
Console.WriteLine(b.get() + b.get());
b.get(); // blocks until some other thread calls put
```
public class Buffer {

    public void give(String s) & async available(int a) {
        if (a == 1) {
            // just one slot available and giving: become full
            full();
        } else {
            // more than one slot available and giving:
            // enable more giving
            available(a - 1);
        }
        // enqueue message for takes
        inBuffer(s);
    }
}
public String take() & async inBuffer(String s) &
    async full() {
    // full and taking: one slot becomes available
    available(1);
    // return message in queue
    return s;
}

public String take() & async inBuffer(String s) &
    async available(int a) {
    // non-full: one more slot becomes available
    available(a + 1);
    // return message in queue
    return s;
}
Producer/Consumer with chords (3)

```java
// constructor
public Buffer(int capacity) {
    available(capacity);
}
```

- Note: there is no guarantee of ordered retrieval
More concurrency models

The Actor Model
The Actor model

• A mathematical model of concurrent computation, introduced by (Hewitt, 1973) and refined by (Agha, 1985) and others
• Actor metaphor: "active agent which plays a role on cue according to a script"
• Process communication through asynchronous message passing
• No shared state between actors
Message-passing concurrency

- Two main means of process communication:
  - *Shared memory*: processes communicate by reading and writing to shared sections of memory
  - *Message-passing*: processes communicate by sending messages to each other
- Synchronous vs. asynchronous:
  - Synchronous: both sender and receiver have to be ready for the communication to take place
  - Asynchronous: the receiver does not have to be ready to receive when the sender sends a message
Actor

• An *actor* is an entity which in response to a message it receives can
  • send finitely many messages to other actors
  • determine new behavior for messages it receives in the future
  • create a finite set of new actors
• Communication via asynchronous message passing
• Recipients of messages are identified by addresses, hence an actor can only communicate with actors whose addresses it has
• A *message* consists of
  • the target to whom the communication is addressed
  • the content of the message
Erlang

• Erlang: functional language, developed by Ericsson since 1986
• Erlang implements the Actor model
• Success story: Ericsson AXD301 switch for telecommunication systems with very high reliability
Erlang syntax for concurrency

- When processes (≈ actors) are created using `spawn`, they are given unique process identifiers (or PIDs)
  \[ \text{PID} = \text{spawn}(	ext{Module, Function, Arguments}) \]
- Messages are sent by passing tuples to a PID with the `!` syntax.
  \[ \text{PID} ! \{ \text{message} \} \]
- Messages are retrieved from the mailbox using the `receive()` function with pattern matching
  ```erlang
  receive
    Message1 -> Actions1 ;
    Message2 -> Actions2 ;
    ...
  end
  ```
Counter example

-module(counter).
-export([start/0,loop/1,increment/1,value/1,stop/1]).

% Interface functions.

start() ->
    spawn(counter, loop, [0]).

increment(Counter) ->
    Counter ! increment.

value(Counter) ->
    Counter ! {self(),value},
    receive
        {Counter,Value} ->
            Value
    end.

% Counter loop.

loop(Val) ->
    receive
        increment ->
            loop(Val + 1);
        {From,value} ->
            From ! {self(),Val},
            loop(Val);
        Other ->
            loop(Val)
    end.