Advances in Programming Languages
APL16: Further language concurrency mechanisms

David Aspinall
(including slides by Ian Stark)

School of Informatics
The University of Edinburgh

Thursday 5 March 2009
Semester 2 Week 8
This is the second of three lectures presenting some programming-language techniques for managing concurrency.

- Basic Java concurrency
- Concurrency abstractions
- Concurrency in some other languages
Outline

1. Concurrency mechanisms

2. Scala

3. Polyphonic C#
1. Concurrency mechanisms
2. Scala
3. Polyphonic C#
Concurrency mechanisms

There is a large design space for concurrent language mechanisms. Two requirements are *separation*, to prevent inconsistent access to shared resources, and *co-operation* for communication between tasks.
Concurrency mechanisms

There is a large design space for concurrent language mechanisms. Two requirements are *separation*, to prevent inconsistent access to shared resources, and *co-operation* for communication between tasks.

Various language paradigms have been followed, e.g.:

- **locks and conditions**: tasks share memory and exclude and signal one another using shared memory (e.g., Java);
- **synchronous message passing**: tasks share communication channels and use *rendezvous* to communicate (e.g., Ada, Concurrent ML);
- **asynchronous message passing**: a task offers a *mail box* which receives messages (e.g., Erlang, Scala Actors);
- **lock-free algorithms or transactional memory**: tasks share memory but detect and repair conflicts.
Concurrency mechanisms

There is a large design space for concurrent language mechanisms. Two requirements are *separation*, to prevent inconsistent access to shared resources, and *co-operation* for communication between tasks.

Various language paradigms have been followed, e.g.:

- **locks and conditions**: tasks share memory and exclude and signal one another using shared memory (e.g., Java);
- **synchronous message passing**: tasks share communication channels and use *rendezvous* to communicate (e.g., Ada, Concurrent ML);
- **asynchronous message passing**: a task offers a *mail box* which receives messages (e.g., Erlang, Scala Actors);
- **lock-free algorithms or transactional memory**: tasks share memory but detect and repair conflicts.

Language designs have also been influenced by mathematical models used to capture and analyse the essence of concurrent systems, for example, *\(\pi\)-calculus*, the *join calculus*, and the *ambient calculus*. 
1 Concurrency mechanisms

2 Scala

3 Polyphonic C#
Scala and Erlang

*Scala* is a functional object-oriented language that compiles to the Java Virtual Machine. It allows full interoperability with Java. Scala is designed by Martin Odersky and his team at EPFL, Lausanne, Switzerland.

Scala’s concurrency is based on the *Actor model* also used in several other languages. A notable commercial success story is Ericsson’s language *Erlang* designed for massively concurrent telecommunications equipment.
Asynchronous message passing

An *actor* is a process abstraction that interacts with other actors by message passing. Message sending is asynchronous. Each actor has a *mailbox* which buffers incoming messages. Messages are processed by matching.

**Sending**

```plaintext
actor ! message

// sender is the last actor
// we received from
sender ! message

// shorthand for above
reply(message)
```

**Receiving**

```plaintext
receive {
  case pattern => action
  ...
  case pattern => action
}
```
Example: ping pong

class Ping(pong: Actor)
  extends Actor {
    def act() {
      var pings = 0;
      pong ! Ping
      while (true) {
        receive {
          case Pong =>
            pong ! Ping
            pings += 1
            if (pings % 1000 == 0)
              Console.println("Ping: pong " + pings)
        }
      }
    }
  }

class Pong extends Actor {
  def act() {
    var pongs = 0
    while (true) {
      receive {
        case Ping =>
          sender ! Pong
          pongs += 1
      }
    }
  }
}

object pingpong
  extends Application {
    val pong = new Pong
    val ping = new Ping(pong)
    ping.start
    pong.start
  }
Actors often take part in sequences of message exchanges, which are more synchronous in nature. There is a special encoding for writing these.

### Sending and receiving

```python
actor !? message

is like

actor ! (self, message)

receive {
    case pattern => ...
}
```
Event-based actors

Actors are either *thread-based* or *event-based*. Thread based actors block on *receive* calls. Event-based actors provide an alternative which uses a more lightweight mechanism.

```
Event based receiving

react {
  case pattern => action
  ...
  case pattern => action
}
```

A *react* statement encapsulates the rest of a computation for an actor and never returns. The event-based framework generates tasks that process messages and suspend and resume actors, using *continuations* derived from the *react* blocks.
class BoundedBuffer[T](N: int) {
  private case class Put(x: T)
  private case object Get
  private case object Stop

  def put(x: T) {
    buffer !? Put(x)
  }

  def get: T =
  (buffer !? Get).asInstanceOf[T]

  def stop() {
    buffer !? Stop
  }
}

private val buffer = actor {
  val buf = new Array[T](N)
  var in = 0; var out = 0; var n = 0
  loop {
    react {
      case Put(x) if n < N =>
        buf(in) = x
        in = (in + 1) % N
        n = n + 1; reply()
      case Get if n > 0 =>
        val r = buf(out)
        out = (out + 1) % N
        n = n - 1; reply(r)
      case Stop => reply()
    }
    exit("stopped")
  }
}
Concurrency in Scala

- Concurrency in Scala is modelled with *actors*.
- Each actor has a *mail box*, to which other threads can send messages asynchronously.
- Actors sift through received messages by *pattern-matching*.
- Scala actors can be either *thread-based* or *event-based*.
- Thread-based actors block JVM threads when waiting to receive a message.
- Event-based actors use task management within a JVM thread to allow cheap context switching between suspended actors.
1. Concurrency mechanisms

2. Scala

3. Polyphonic C#
Polyphonic C# is a mild extension of C# which introduced novel primitives for writing concurrent programs, based on the join calculus. The design was inspired by these ideas:

- Focus on communication rather than concurrency.
- Unify message passing with method invocation.
- Look not just for individual messages but patterns of messages.

Polyphonic C# itself is no longer maintained. The concurrency mechanisms also appear in Cω, and are provided in the Joins library for C# other .NET languages. Similar notions have been applied in JoCaml and Join Java.
Asynchronous methods

Conventional method invocation in C# is *synchronous*: when code calls a method on an object, it cannot continue until that method completes.

In contrast, when code invokes an *asynchronous* method, it continues at once, and does not have to wait for the method to finish.

Chords

Standard method declarations associate one piece of code (the *body*) to each method name (up to overloading by parameter type and number).

In Polyphonic C#, a *chord* declares code that is to be executed only when a particular combination of methods are invoked.
**public class** Buffer {

**public** String get() **&** **public** async put(String s) {
    return s;
}
}

This has two methods, `get` and `put`, jointly defined in a *chord*, with a single return statement in the body.

Consumers call `get()`: this blocks until a producer invokes `put(s)`, and then the chord is complete so `s` is returned to the consumer.

Producers call `put(s)`: if a consumer is waiting on `get()`, then the chord is complete and value is handed on; if not, the call is noted, and control returns to the producer. Either way, the `async` call returns at once.

Multiple `put` or `get` calls can be outstanding at any time.
Example: unbounded concurrent buffer

```java
public class Buffer {

    public String get() & public async put(String s) { return s; }

}
```

- No threads are spawned: the body of the chord is executed by the caller of the synchronous `get` method.
- Where there are multiple threads, it is entirely thread-safe: several producers and consumers can run simultaneously.
- No critical sections, monitors or mutual exclusion: there is no shared storage for interference.
- No explicit locks: the compiler looks after the brief locking required at the moment of chord selection.
Example: unbounded concurrent buffer

```java
public class Buffer {

    public String get() & public async put(String s) { return s; }

}
```

- Each chord may combine many method names.
- At most one method in a chord can be synchronous.
- Each method can appear in multiple chords.
- A chord may be entirely asynchronous.
- Synchronous calls may block; asynchronous calls return at once.
- Calls stack up until a chord is matched.
Example: one-place buffer

```java
public class OnePlaceBuffer {

    public OnePlaceBuffer() { empty(); }

    public void put(String s) & private async empty() {
        contains(s);
        return;
    }

    public String get() & private async contains(String s) {
        empty();
        return s;
    }
}
```
Workings of the one-place buffer

The class has four methods:

- Two public synchronous methods `put(s)` and `get()`;
- Two private asynchronous methods `empty()` and `contains(s)`.  

There is always exactly one `empty()` or `contains(s)` call pending. No threads are needed, but where there is concurrency the code remains safe.

- Method `put(s)` blocks unless and until there is an `empty()` call.
- Method `get()` blocks unless and until there is a `contains(s)` call.

The code operates a simple state machine:
Polyphonic C# Concurrency

- Central notion of *asynchronous* computation.
- Implicit concurrency: no explicit threads, locks, mailboxes, channels, ...; although all these could be coded up.
- Synchronization points made explicit with *chords*.
- Declarative presentation makes compiler optimisations possible.