Advances in Programming Languages

APL14: Polyphonic C#

Ian Stark

School of Informatics
The University of Edinburgh

Monday 3 March 2008
Semester 2 Week 9

Programming-Language Techniques for Concurrency

This is the second of three lectures presenting some programming-language techniques for managing concurrency.

- Java, Erlang
- Polyphonic C#
- Cautionary Tales

Programming-Language Techniques for Concurrency

This is the second of three lectures presenting some programming-language techniques for managing concurrency.

- Java, Erlang
- Polyphonic C#
- Cautionary Tales

Concurrent Programming

"It is a truth universally acknowledged that concurrent programming is significantly more difficult than sequential programming"

[Cω language overview]

Concurrency is useful:

- Efficient use of mixed resources (disk, memory, network)
- Responsiveness (GUI, hardware interrupts, managing mixed resources)
- Speed (multiprocessing, hyperthreading, multicore)
- Multiple clients (database engine, web server)

Concurrency is hard:

- Interference (shared store, simultaneous modification)
- Liveness (deadlock, livelock, lack of progress)
- Fairness (scheduling, prioritization, starvation)
- Safety (correctness, error handling, specification)

Concurrency in Java and C#

Java

. . . has language facilities for spawning a new java.lang.Thread, with **synchronized** code blocks using per-object locks to protect shared data, and signalling between threads with wait and notify.

C#

. . . has language facilities for spawning a new System.Threading.Thread, to **lock** code blocks using per-object locks to protect shared data, and signalling between threads with Wait and Pulse methods.

More Language Concurrency

There are many ways to program concurrency, such as the following from functional languages:

Erlang: Multiple share-nothing threads, each with a single mailbox for asynchronous communication by message-passing.

Concurrent ML: Multiple threads, multiple channels for synchronous message-passing communication between them.

Concurrent Haskell: Multiple threads, asynchronous communication through MVar mutable variables.

There are also mathematical models to capture and analyse the essence of these concurrent systems: such as the π-*calculus*, the *join-calculus*, and the *ambient calculus*.

Towards Yet Another Concurrency Model

Looking for ways to improve concurrent programming in object-oriented languages, consider the following themes:

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

Polyphonic C#

Polyphonic C# is a mild extension of C# which provides novel primitives for writing concurrent programs, based on the join calculus.

> "The language presents a simple and powerful model of concurrency which is applicable both to multithreaded applications running on a single machine and to the orchestration of asynchronous, event-based applications communicating over a wide area network."
>
> [Benton, Cardelli, Fournet]

These extensions also appear in $\mathrm{C}\omega$, the research programming language we met earlier as the source of LINQ.

Polyphony, *n.* **1. a.** *Music.* Harmony; *esp.* the simultaneous and harmonious combination of a number of individual melodic lines.
Oxford English Dictionary, draft revision, June 2007

New Constructions in Polyphonic C#

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.

Example: Storage Cell

The following C# code defines a straightforward storage cell, containing a single String value.

```
public class Cell {

    private String contents = "";

    public String get() { return contents; }

    public void put(String s) { contents=s; }
}
```

Likely to be thread safe, provided put and get methods remain this simple.

Example: Storage Cell

The following C# code defines a straightforward storage cell, containing a single String value.

```
public class Cell {

    private String contents = "";

    public String get() { return contents; }

    public async put(String s) { contents=s; }
}
```

The *asynchronous* put method now returns immediately to its caller, and may use a separate thread to update the cell contents.

Example: Unbounded Concurrent Buffer

```
public class Buffer {

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

This still has two methods, get and put, but now 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

```
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

```
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 always return at once.
- Calls stack up until a chord is matched.

Example: One-Place Buffer

```
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

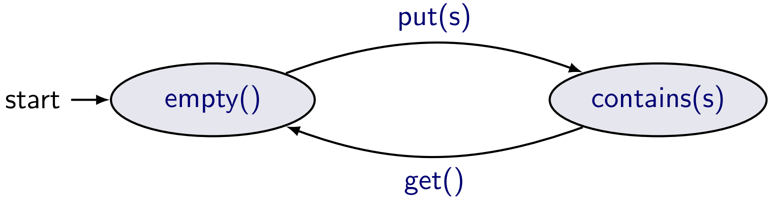
The class has four methods:

- 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:



Example: Callbacks and Distribution

```
delegate async IntCallback(int value); // Declare function type

class Service {
    public async request(String arg, IntCallback c) {
        int result;
        ...                // Compute result in some interesting way
        c(result);         // Pass it to asynchronous callback
        ...                // Tidy up
    }
}
```

Client code can dispatch a request to a Service, do some work of its own, then rendezvous to pick up the result when ready.

Compare XMLHttpRequest from Javascript, used in AJAX for asynchronous communication with web services.

Example: Callbacks and Distribution

```
delegate async IntCallback(int value); // Declare function type

class Service {
    public async request(String arg, IntCallback c) {
        int result;
        ...             // Compute result in some interesting way
        c(result);      // Pass it to asynchronous callback
        ...             // Tidy up
    }
}
```

Several requests can be dispatched together, and a client might wait until all or any of them are completed.

The Service and client can be on different machines: asynchronous request and callback methods means that they distribute well.

Other Examples

Other classic concurrency idioms have versions in Polyphonic C#:

- Combining shared and exclusive access to resources with multiple-readers / single-writer (just five chords).
- Locks, semaphores, condition variables (if you want them).
- Active objects, concurrent objects, Actors.
- Concurrent publish/subscribe, subject/observer pattern.
- Custom schedulers: thread pooling, worker threads.
- ⟨ *insert your favourite concurrent programming problem* ⟩

Some of these are just to show that chords are as expressive as other paradigms: in actual use, the ideal is to raise the level of abstraction and avoid explicit concurrency management.

Features of Polyphonic C#

- Central notion of *asynchronous* computation, directly addressing application responsiveness.
- Concurrency is implicit: no explicit threads, locks, mailboxes, channels,. . . ; although all these could be coded up.
- High-level description of the desired interaction profile through *chords*.
- Declarative presentation means that a compiler can transform and optimize as appropriate: to avoid thread spawning if not required; reusing existing threads; worker threads, thread pools.
- Transformation could in principle also include platform details: multiple processors, multicore, distributed client/server.

The same notions have been applied to other languages in *JoCaml* and *Join Java*, and are also available in the *Joins* library for C# and VB.NET.

Summary

- Java and C# use explicit shared-memory concurrency; with threads, locks, monitors and semaphores.
- Erlang has explicit processes but they share no store, instead communicating by mailboxes.
- This can be generalised (Concurrent ML, Concurrent Haskell, . . .) to multiple threads and multiple named communication channels.
- Polyphonic C# / Cω provide *asynchronous methods* and *chords* to orchestrate implicitly concurrent behaviour.

Homework

Read two pieces describing concurrent programming in Polyphonic C#:

- The three-page online introduction at http://research.microsoft.com/~nick/polyphony/intro.htm
- Nick Benton. Jingle Bells: Solving the Santa Claus Problem in Polyphonic C#. http://research.microsoft.com/~nick/polyphony/santa.pdf

If you are interested, find out more in the concurrency sections of the Cω web site, and in the following paper:

James Gosling, Bill Joy, Guy Steele.
The Java Language Specification (First Edition), Chapter 17.
Addison-Wesley, 1996.
Modern Concurrency Abstractions for C#.
ACM Transactions on Programming Languages and Systems
26(5):269–804. September 2004. DOI: 10.1145/1018203.1018205