Modern Concurrency Abstractions for C$^\sharp$

Nick Benton Luca Cardelli Cédric Fournet

Microsoft Research
Cambridge, UK
{nick,luca,fournet}@microsoft.com

Abstract

Polyphonic C$^\sharp$ is an extension of the C$^\sharp$ language with new asynchronous concurrency constructs, based on the Join Calculus. We describe the design and implementation of the language and give examples of its use in addressing a range of concurrent programming problems.

1 Introduction

1.1 Languages and Concurrency

Concurrency is an important factor in the behavior and performance of modern code: concurrent programs are difficult to design, write, reason about, debug, and tune. Concurrency can significantly affect the meaning of virtually every other construct in the language (beginning with the atomicity of assignment), and can affect the ability to invoke libraries. Yet, most popular programming languages treat concurrency not as a language feature, but as a collection of external libraries that are often underspecified.

Considerable attention has been given, after the fact, to the specification of important concurrency libraries [4, 12, 11, 7] to the point where one can usually determine what their behavior should be under any implementation. Yet, even when the concurrency libraries are satisfactorily specified, the simple fact that they are libraries, and not features of the language, has undesirable consequences.

Many features can be provided, in principle, either as language features or as libraries: typical examples are memory management and exceptions. The advantage of having such features "in the language" is that the compiler can analyze them, and can therefore produce better code and warn programmers of potential and actual problems. In particular, the compiler can check for syntactically embedded invariants which would be difficult to extract from a collection of library calls. Moreover, programmers can more reliably state their intentions through a clear syntax, and tools other than the compiler can more easily determine the programmers' intentions. Domain Specific Languages [25, 16] are an extreme example of this linguistic approach: new ad-hoc languages are routinely proposed not to replace general-purpose language, but to facilitate domain-specific code analysis by the simple fact of expressing domain-related features as primitive language constructs.

We believe that concurrency should be a language feature and a part of language specifications. Serious attempts in this direction were made beginning in the 1970's with the concept of monitors [14] and the Occam language [21] (based on Hoare's Communicating Sequential Processes [15]). The general notion of monitors has become very popular, particularly in its current object-oriented form of threads and object-bound mutexes, but it has been provided at most as a veneer of syntactic sugar for optionally locking objects on method calls.

Many things have changed in concurrency since monitors were introduced. Communication has become more asynchronous, and concurrent computations have to be "orchestrated" on a larger scale. The concern is not as much in the efficient implementation and use of locks on a single processor or multiprocessor, but on the ability to handle asynchronous events without unnecessarily blocking clients for long periods, and without deadlocking. In other words, the concern is shifting from shared-memory concurrency to message/event-oriented concurrency.

These new requirements deserve programming constructs that can handle well asynchronous communications and that are not shackled to the shared-memory approach. Despite the development of a large collection of design patterns [20] and of many concurrent languages [2, 24, 1], only monitors have gained widespread acceptance as programming constructs.

An interesting new linguistic approach has emerged recently with Fournet and Gonthier's Join Calculus [9, 8], a process calculus well-suited to direct implementation in a distributed setting. Other languages, such as JoCaml [6] and Funnel [23], combine similar ideas with the functional programming model. Here we propose an adaptation of Join Calculus ideas to an object-oriented language that already has an existing threads-and-locks concurrency model.

1.2 Asynchronous Programming

Asynchronous events and message passing are increasingly used at all levels of software systems. At the lowest level, device drivers have to respond promptly to asynchronous device events, while being parsimonious on resource use. At the Graphical User Interface level, code and programming models are notoriously complex because of the asynchronous nature of user events; at the same time, users hate being blocked unnecessarily. At the wide-area network level, e.g. in collaborative applications, distributed workflow or web services, we are now experiencing similar problems and complexity because of the asynchronous nature and latencies of global communication.

All these areas naturally lead to situations where there are many asynchronous messages to be handled concurrently, and where many threads are used to handle them. Threads are still an expensive resource on most systems. However, if we can somewhat hide the use of messages and threads behind a language mechanism, then many options become possible. A compiler may transform some patterns of concurrency into state machines, optimize the use of queues, use lightweight threads when possible, avoid forking threads when not necessary, and use thread pools. All this is really possible only if one has a handle on the spectrum of "things that can happen": this handle can be given by a syntax for concurrent operations that can both hide and enable multiple implementation techniques.

Therefore, we aim to promote abstractions for asynchronous programming that are both high-level, from the point of view of a programmer, and that enable multiple low-level optimizations, from the point of view of a compiler and run-time systems. We propose an extension of the C♯ language with modern concurrency abstraction for asynchronous programming. In tune with the musical spirit of C♯ and with the "orchestration" of concurrent activities, we call this language Polyphonic C♯.[1]

1.3 C♯ and .NET

C♯ is a modern, type-safe, object-oriented programming language recently introduced by Microsoft as part of Visual Studio.NET [13]. C♯ programs run on top of the .NET Framework, which includes a multilanguage execution engine and a rich collection of class libraries.

The .NET execution engine provides a multithreaded execution environment with synchronization based on locks attached to each heap-allocated object. The C♯ language includes a **lock** statement, which obtains the mutex associated with a given object during the execution of a block. In addition, the .NET libraries implement many traditional concurrency control primitives such as semaphores, mutexes and reader/writer locks, as well as an asynchronous programming model based on delegates.[2] The .NET Framework also provides higher-level infrastructure for building distributed applications and services, such as SOAP-based messaging and remote method call.

The concurrency and distribution mechanisms of the .NET Framework are powerful, but they are also undeniably complex. Quite apart from the bewildering array of primitives which are more or less 'baked in' to the infrastructure, there is something of a mismatch between the 1970s model of concurrency on a single machine (shared memory, threads, synchronization based on mutual exclusion) and the asynchronous, message-based style which one uses for programming web-based applications and services. C♯ therefore seems an ideal testbed for our ideas on language support for concurrency in mainstream languages.

2 Polyphonic C♯ Language Overview

This section introduces the new constructs in Polyphonic C♯ and their use in basic concurrent programming.

[1] *Polyphony* is musical composition that uses simultaneous, largely independent, melodic parts, lines, or voices (Encarta World English Dictionary, Microsoft Corporation, 2001).

[2] An instance of a delegate class encapsulates an object and a method on that object with a particular signature. So a delegate is more than a C-style function pointer, but slightly less than a closure.

2.1 The Basic Idea

To C♯'s fairly conventional object-oriented programming model, Polyphonic C♯ adds just two new concepts: *Asynchronous methods* and *Chords*.

2.1.1 Asynchronous Methods

Conventional methods are synchronous, in the sense that the caller makes no progress until the callee completes. In Polyphonic C♯, if a method is declared *asynchronous* then any call to it is guaranteed to return (essentially) immediately. Asynchronous methods never return a result and are declared by using the **async** keyword instead of **void**. Calling an asynchronous method is much like sending a message, or posting an event.

Since asynchronous methods have to return immediately, the behavior of a method such as

```
async postEvent(EventInfo data) {
  // large method body
}
```

is pretty much the only thing it could be: the call returns immediately and 'large method body' is scheduled for execution in a different thread (either a new one spawned to service this call, or a worker from some pool). However, this kind of definition is actually rather rare in Polyphonic C♯. More commonly, asynchronous methods are defined using chords, as described below, and do not necessarily require new threads.

2.1.2 Chords

A *chord* (also called a 'synchronization pattern', or 'join pattern') consists of a header and a body. The header is a set of method declarations separated by &. The body is only executed once *all* the methods in the header have been called. Method calls are implicitly queued up until/unless there is a matching chord. Consider for example

```
class Buffer {
  string get() & async put(string s) {
    return s;
  }
}
```

The code above defines a class *Buffer* declaring two instance methods which are defined together in a single chord. The first method **string** *get*() is a synchronous method taking no arguments and returning a **string**. The second method **async** *put*(**string** *s*) is asynchronous (so returns no result) and takes a **string** argument.

If *buff* is a instance of *Buffer* and one calls the synchronous method *buff.get*() then there are two possibilities:

- If there has previously been an unmatched call to *buff.put*(*s*) (for some string *s*) then there is now a match, so the pending *put*(*s*) is de-queued and the body of the chord runs, returning *s* to the caller of *buff.get*().
- If there are no previous unmatched calls to *buff.put*(_) then the call to *buff.get*() blocks until another thread supplies a matching *put*(_).

Conversely, on a call to the asynchronous method *buff.put*(*s*), the caller will never wait but there are two possible behaviours with regard to other threads:

- If there has previously been an unmatched call to *buff.get*() then there is now a match, so the pending call is de-queued and its associated blocked thread is reawakened to run the body of the chord, which will return s.
- If there are no pending calls to *buff.get*() then the call to *buff.put*(s) is simply queued up until one arrives.

Exactly *which* pairs of calls will be matched up is unspecified, so even a single-threaded program such as

```
Buffer buff = new Buffer();
buff.put("blue");
buff.put("sky");
Console.Write(buff.get() + buff.get());
```

is non-deterministic (printing either "bluesky" or "skyblue").[3]

Note that the implementation of a *Buffer* does not involve spawning any threads – whenever the body of the chord runs, it does so in a preexisting thread (viz. the one which called *get*()). The reader may at this point wonder what the rules are for deciding in which thread a body runs, or how we know to which method call the final value computed by the body will be returned. The answer is that in any given chord, at most one method may be synchronous. If there is such a method, then the body runs in the thread associated with, and the value is returned to, the call to that method. If there is no such method (i.e. all the methods in the chord are asynchronous) then the body runs in a new thread and there is no value to return.

It should also be pointed out that the *Buffer* code, trivial though it is, is completely thread-safe. The locking that is required (for example to prevent the argument to a single *put* being returned to two distinct *get*s) is generated automatically by the compiler.

The *Buffer* example uses a single chord to define two methods. It is also possible (and common) to have multiple chords involving a given method. For example:

```
class Buffer {
  string get() & async put(string s) {
    return s;
  }

  string get() & async put(int n) {
    return n.ToString();
  }
}
```

Now we have defined two methods for putting things into the buffer (which happen to be distinguished by type rather than name) and a single method for getting things out. A call to *get*() can synchronize with a call to either of the *put*() methods. If there are pending calls to both *put*()s, then which one synchronizes with a subsequent *get*() is unspecified.

2.2 Programming in Polyphonic C♯

2.2.1 A Simple Cell Class

We start with an implementation of a simple one-place cell class. Cells have two public synchronous methods: **void** *put*(*Object o*) and *Object get*(). A call to *put* blocks until the cell is empty and then fills the cell with its argument. A call to *get* blocks until the cell is full and then removes and returns its contents:

```
class OneCell {
  public OneCell() {
    empty();
  }

  public void put(Object o) & async empty() {
    contains(o);
  }

  public Object get() & async contains(Object o) {
    empty();
    return o;
  }
}
```

In addition to the two public methods, the class uses two private asynchronous methods, *empty*() and *contains*(*Object o*), to carry the state of cells. There is a simple declarative reading of the constructor and the two chords which explains how this works:

constructor: When a cell is created, it is initially *empty*().

***put*-chord:** If we *put* an *Object o* into a cell which is *empty*() then the cell *contains*(o).

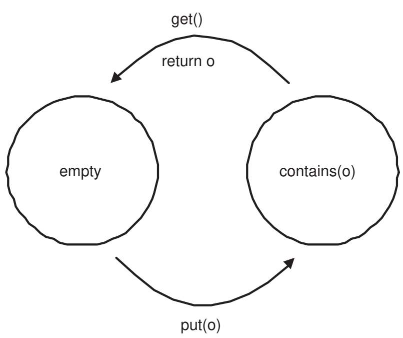
***get*-chord:** If we *get*() the contents of a cell which *contains* an *Object o* then afterwards the cell is *empty*() and the returned value is o.

implicitly: In all other cases, *put*s and *get*s wait.

The technique of using private asynchronous methods (rather than fields) to carry state is very common in Polyphonic C♯. Observe that the constructor establishes, and every body in class *OneCell* preserves, a simple and easily verified invariant:

> There is always exactly one pending asynchronous method call: either an *empty*() or a *contains*(o), for some *Object o*.

(In contrast there may be an arbitrary number of client threads blocked with pending calls to *put* or *get*, or even concurrently running statement **return** o within the last body.) Hence one can also read the class definition as a direct specification of an automaton:



[3] Of course, in any real implementation the nondeterminism in this very simple example will be resolved statically, so different executions will always produce the same result, but this is not part of the official semantics.

2.2.2 Reader-Writer Locks

As a more realistic example of the use of asynchronous methods to carry state and chords to synchronize access to that state, we now consider the classic problem of protecting a shared mutable resource with a multiple-reader, single-writer lock. Clients each request and then release either shared access or exclusive access, using the corresponding public methods *Shared*, *ReleaseShared*, *Exclusive*, and *ReleaseExclusive*. Requests for shared access block until no other client has exclusive access, whilst requests for exclusive access block until no other client has any access. A canonical solution to this problem using traditional concurrency primitives in Modula 3 may be found in [3]; using Polyphonic C$^\sharp$, it can be written with just five chords:

```
class ReaderWriter
{
  ReaderWriter() { Idle(); }

  public void Shared() & async Idle() { S(1); }
  public void Shared() & async S(int n) { S(n+1); }
  public void ReleaseShared() & async S(int n) {
    if (n == 1) Idle(); else S(n−1);
  }
  public void Exclusive() & async Idle() {}
  public void ReleaseExclusive() { Idle(); }
}
```

Provided that every release follows the corresponding request, the invariant is that the state of the lock (no message, a single message *Idle*(), or a single message *Shared*(n) with $n > 0$) matches the kind and number of threads currently holding the lock (an exclusive thread, no thread, or n sharing threads).

It is a matter of choice whether to use private fields or parameters in private messages. In the example above, n makes sense only when there is an S message present. Nonetheless, we could write instead the following equivalent code:

```
class ReaderWriterPrivate
{
  ReaderWriter() { Idle(); }
  private int n; // protected by S()

  public void Shared() & async Idle() { n=1; S(); }
  public void Shared() & async S() { n++; S(); }
  public void ReleaseShared() & async S() {
    if (−−n == 0) Idle(); else S();
  }
  public void Exclusive() & async Idle() {}
  public void ReleaseExclusive() { Idle(); }
}
```

Our model of concurrency provides basic fairness properties. In cases when some application-specific fairness is required, one can supplement it with programmed fairness. For instance, we could further refine our code to implement some fairness between readers and writers, by adding extra shared states, T(), when we don't accept new readers, and *IdleExclusive*(), when we provide the exclusive lock to a previously-selected thread.

```
class ReaderWriterFair
{
  ... // same content as above, plus:
  public void ReleaseShared() & async T() {
    if (−−n == 0) IdleExclusive(); else T();
  }
  public void Exclusive() & async S() { T(); wait(); }
  void wait() & async IdleExclusive() {}
}
```

2.2.3 Active Objects

A common pattern in other concurrent object oriented languages is that of *active objects*. These have an independent thread of control associated with each instance which is used to process (typically sequentially) messages sent (typically asynchronously) from other objects. One way to express this pattern in Polyphonic C$^\sharp$ is via inheritance from an abstract base class:

```
public abstract class ActiveObject {
  protected bool done;

  abstract protected void ProcessMessage();

  public ActiveObject () {
    done = false;
    mainLoop();
  }

  async mainLoop() {
    while (!done) {
      ProcessMessage();
    }
  }
}
```

The constructor of *ActiveObject* calls the asynchronous method *mainLoop*() which spawns a new message-handling thread for that object. Subclasses of *ActiveObject* then define chords for each message to synchronize with a call to *ProcessMessage*(). Here, for example, is a skeleton of an active object which multicasts stock quote messages to a list of clients:

```
public class StockServer : ActiveObject {
  private ArrayList clients;

  public async AddClient(Client c) // add new client
  & void ProcessMessage() {
    clients.Add(c);
  }

  public async WireQuote(Quote q) // new quote off wire
  & void ProcessMessage() {
    foreach (Client c in clients) {
      c.UpdateQuote(q);  // and send to all  clients
    }
  }

  public async CloseDown() // request to terminate
  & void ProcessMessage() {
    done = true;
  }
}
```

Interestingly, in this example one cannot move the *CloseDown*() chord to the superclass (to share it amongst all

ActiveObjects) since that would violate a restriction on combining overriding with synchronization, which we discuss in the next section.

2.2.4 Custom Schedulers

In Polyphonic C♯, we have to both coexist with and build upon the existing threading model. Because these threads are relatively expensive, and are the holders of particular locks, C♯ programmers often need explicit control over thread usage. In such cases, Polyphonic C♯ is a convenient way to write what amount to custom schedulers for a particular application.

To illustrate this point, we present an example in which we dynamically schedule series of related calls in large batches, to favour locality. (This is loosely related to what is sometimes called 'staged' or 'pipelined' computation [18].) The two following classes model such batch computations, represented as *Heavy* objects that have large startup costs and limited concurrency. Pragmatically, those costs may be due to a large code and data footprint. The helper class *Token* enables us to limit the number of active *Heavy* objects, here 2.

```
class Token {
    public void Grab() & public async Free() {}
    public Token(int n) { for (; n-- > 0 ;) Free(); }
}
class Heavy {
    static Token tk = new Token(2); // limits parallelism
    public Heavy (int q) { tk.Grab (); ...; }  // rather slow
    public int Work(int p) { return ...; } // rather fast
    public void Close() { tk.Free(); }
}
```

The class below implements our scheduler. To each task, *Burst* provides a front-end that attempts to organise calls into long series that share the startup cost. A burst can be in two states, represented by either *idle*() or *open*(). The state is initially idle. When a first thread tries to access the resource, the state becomes open, then this thread proceeds with the potentially-blocking *Heavy*(q) call. As long as the state is open, subsequent callers are queued-up. When the first thread completes its own task, and before releasing the *Heavy* resource, the first thread also processes the tasks for all pending calls and resumes their respective threads with the results. Meanwhile, the state is still open, and new threads may be queued-up, so the process is repeated until no other thread is present. Eventually, the state becomes idle again. The helper class *Thunk* is used to block each queued-up thread and resume it with the result r, in asynchronous message-passing style.

```
class Burst {
    int others = 0; int q;
    public Burst(int q) { this.q = q; idle (); }

    public int Work(int p) & async idle() {
        open();
        Heavy h = new Heavy(q);
        int r = h.Work(p);
        helpful(h); // any delayed threads?
        h.Close();
        return r;
    }
    public int Work(int p) & async open() {
        others++; open();
        Thunk t = new Thunk(); delayed(t,p);
        return t.Wait(); // usually blocking
    }
    void helpful(Heavy h) & async open() {
        if (others == 0) idle();
        else {
            int batch = others; others = 0;
            open();
            while(batch-- > 0) extraWork(h);
            helpful(h); // newly-delayed threads?
        }
    }
    void extraWork(Heavy h) &
    async delayed(Thunk t,int p) {
        t.Done(h.Work(p));}
}
class Thunk {
    public int Wait() & public async Done(int r) {
        return r;
    }
}
```

We omit the code that allocates an array of *Burst* objects to be shared by all threads, and some performance test code, which unsurprisingly exhibits a large speedup when concurrent threads call *Burst* rather than directly calling *Heavy*.

3 Informal Specification

3.1 Grammar

The syntactic extensions to the C♯ grammar [13, Appendix C] are very minor. We add a new keyword **async**, and in *method-header*s we allow **async** instead of a return type. In *class-member-declaration*s, we replace *method-declaration* with *chord-declaration*:

chord-declaration ::=
  method-header [& *method-header*]* *body*
method-header ::=
  attributes modifiers return-type member-name(*formals*)
return-type ::=
  type | **void** | **async**

We call a chord declaration *trivial* if it declares a single, synchronous method (i.e. it is a standard C♯ method declaration).

3.2 Well-Formedness

Extended classes are subject to a number of specific well-formedness conditions:

- In a single *method-header*:

  1. If *return-type* is **async** then the formal parameter list *formals* may not contain any **ref** or **out** parameter modifier.[4]

- In a single *chord-declaration*:

  2. At most one *method-header* may have a non-**async** *return-type*.

[4]Neither **ref** nor **out** parameters are safe for asynchronous messages, since they are both passed as addresses of locals in a stack frame which may have disappeared when the message is processed.

3. If the chord has a *method-header* with *return-type type*, then *body* may use **return** statements with *type* expressions, otherwise *body* may use empty **return** statements.

4. All the *formals* appearing in *method-headers* must have distinct identifiers.

5. Two *method-headers* may not have both the same *member-name* and the same argument type signature.

6. The *method-headers* must either all declare instance methods or all declare static methods.

- Within a particular class:

7. All *method-headers* with the same *member-name* and argument type signature must have the same *return-type* and identical sets of *attributes* and *modifiers*.

8. If it is a value class (**struct**), then only static methods may appear in non-trivial chords.

9. If any *chord-declaration* includes a virtual method m with the **override** modifier[5], then any method m' which *in the superclass containing the overridden definition of* m appears in a chord with m must also be overridden in the subclass.

Most of these conditions are fairly straighforward, though Condition 9, which provides a conservative, but simple, sanity check when refining a class that contains chords, deserves some further comment.

In general, implementation inheritance and concurrency do not mix well [22]. Our approach is to enforce a separation of these two concerns: a series of chords must be syntactically local to a class or a subclass declaration; when methods are overriden, all their chords must also be completely overriden. If one takes the view that the implementation of a given method consists of all the sychronization and bodies of all the chords in which it appears then the our inheritance restriction seems not unreasonable, since in (illegal) code such as

```
class C {
   virtual void f() & virtual async g() { /* body1 */ }
   virtual void f() & virtual async h() { /* body2 */ }
}

class D : C {
   override async g() { /* body3 */ }
}
```

one would, by overriding $g()$, have also 'half' overridden $f()$.

More pragmatically, removing the restriction on inheritance makes it all too easy to introduce inadvertent deadlock (or 'async leakage'). If the above code were legal, then code written to expect instances of class C which makes matching calls to $f()$ and $g()$ would fail to work when passed an instance of D – all the calls to $g()$ would cause *body3* to run and all the calls to $f()$ would deadlock.

Note that the inheritance restriction means that code such as

```
class C {
   virtual void f() & private async g() { /* body1 */ }
}
```

is incorrect: declaring just one of $f()$ and $g()$ to be **virtual** makes no sense, as overriding one requires the other to be overriden too. It is also worth observing that there is a transitive closure operation implicit in our inheritance restriction: if $f()$ is overridden and joined with $g()$ then because $g()$ must be overridden, so must any method $h()$ which is joined with $g()$ and so on.

It is possible to devise more complex and permissive rules for overriding which would, for example, allow the earlier *ActiveObject* example to implement the *CloseDown* chord in the base class. Our current rule has the advantage of simplicity, but we refer the reader to [10] for a more thorough study of inheritance in the Join calculus, including more advanced type systems for its control.

The second restriction above (that there be at most one synchronous method in a chord) is also caused by a potentially bad interaction between existing C♯ features and the pure Join calculus. Allowing more than one synchronous call to appear in a single chord would give a potentially useful *rendez-vous* facility (provided one also added syntax allowing results to be returned to particular calls). However, one would then have to decide in which of the blocked threads the body ran, and this decision is observable to the C♯ programmer. If this were simply because thread identities can be obtained and checked for equality, the problem would be fairly academic (object-oriented programmers typically not being especially worried about strict observational equivalences). However, since reentrant locks are associated with threads, the choice of thread could make a significant difference to the synchronization behaviour of the program, thus making & 'very' non-commutative.

Of course, it is not hard to program a rendez-vous explicitly, though asymmetrically, in Polyphonic C♯. In the following example, calls from different threads of the methods f and g will wait for each other and then exchange arguments before proceeding. We use an auxiliary class C for blocking and resuming the g thread:

```
class C {
  int wait() & async reply(int i) { return i; }
}
int g(int j) {
  C c = new C(); a(j,c); return c.wait();
}
int f(int i) & private async a(int j,C c) {
  c.reply(i); return j;
}
```

3.3 Typing Issues

We treat **async** as a subtype of **void** and allow 'covariant return types' just in the case of these two (pseudo)types. Thus

- an **async** method may override a **void** one,
- a **void** delegate may be created from an **async** method, and
- an **async** method may implement a **void** method in an interface

[5]In C♯, methods which are intended to be overridable in subclasses are explicitly marked as such by use of the **virtual** modifier, whilst methods which are intended to override ones inherited from a superclass must explicitly say so with the **override** modifier.

but not conversely. This design makes intuitive sense (an **async** method *is* a **void** one, but has the extra property of returning 'immediately') and also maximises compatibility with existing code (superclasses, interfaces and delegate definitions) which makes use of **void**.

4 Implementation

This section describes the implementation of chords using lower-level concurrency primitives. The compilation process is best explained as a translation from a polyphonic class to a plain C♯ class. The resulting class has the same name and signature as the source class, and also has private state and methods to deal with synchronization.

4.1 Synchronization and State Automata

In the implementation of a polyphonic class, each method body combines two kinds of code, corresponding to the synchronization of polyphonic method calls (generated from the chord headers) and to their actual computation (copied from the chord bodies), respectively.

We now describe how the synchronization code is generated from a set of chords. Since synchronization is statically defined by those chords, we can efficiently compile it down to a state automaton. This is the approach initially described in [19], though our implementation does not construct explicit state machines.

The *synchronization state* consists of pending calls for any method that occurs in a chord, that is, threads for regular methods and messages for asynchronous methods. However, synchronization effectively depends on a much simpler state that records only the presence of pending calls; the actual parameters and the calling contexts become relevant only after a chord is fired. Hence, the whole synchronization state can be summarized in a word, with a single bit that records the presence of (one or more) pending calls, for every method appearing in a least a chord. Accordingly, every chord declaration is represented as a constant word with a bit set for every method appearing in that chord, and the synchronization code checks whether a chord can be fired by comparing the synchronization word with these precomputed bitmasks.

4.1.1 Performance considerations

The cost of polyphonic method calls should be similar to the cost of regular method calls unless a synchronized method call blocks waiting for **async** messages—in that case, we cannot avoid paying the rather high cost of dynamic thread scheduling.

When an asynchronous method is called, it performs a bounded amount of computation on the caller thread before returning.

When a regular, synchronized method is called, the critical path to optimize is the one in which, for at least one chord, all complementary asynchronous messages are already present. In that case, the synchronization code retrieves the content of the complementary messages, updates the synchronization state, and immediately proceeds with the method body. Conversely, when there is no such chord, the thread must be suspended, and the cost of running our synchronization code is likely to be small as compared to lower-level context-switching and scheduling.

Firing a completely asynchronous chord is always comparatively expensive since it involves spawning a new thread. Hence, when an asynchronous message arrives, it makes sense to check for matches with synchronous chords first. We also lower the cost of asynchronous chords by using .NET's *thread pool* mechanism rather than simply spawning a fresh operating system thread every time. The scheduling policy of the thread pool is not optimal for all applications, however, so we intend to use attributes to allow programmer control over thread creation policy.

4.1.2 Low-level Concurrency

The code handling the chords must be unconditionally thread-safe, for all source code in the class. To this end, we use a single, auxiliary lock to protect our private synchronization state. (We actually use the regular object lock for one of the queues.) Locking occurs only for short period of times, for each incoming call that goes through the chords, so hopefully the lock will nearly always be available.

This lock is independent of the regular object lock, which may be used as usual to protect the rest of the state and prevent race conditions.

4.2 The Translation

We now present, by means of a simple example, the details of the translation of Polyphonic C♯ into ordinary C♯. The translation presented here is actually an abstraction of that which we have implemented. For didactic purposes, we modularise the translated code by introducing auxiliary classes for queues and bitmasks, whereas our current implementation essentially inlines the code contained in these classes.

4.2.1 Supporting Classes

The following value class (structure) provides operations on bitmasks:

```
struct BitMask {
  private int v; // = 0;
  public void set(int m) { v |= m; }
  public void clear(int m) { v &= ~m; }
  public bool match(int m) { return (~v & m)==0; }
}
```

Next, we define the classes that represent message queues. To every asynchronous method, the compiler associates a message-queue that stores pending messages for that method, with an *empty* property for testing its state and two methods *add* and *get* for adding an element to the queue and getting an element back (when asserting that the queue is not empty). The implementation of each queue depends on the message contents (and, potentially, on compiler-deduced invariants); it does not necessarily use an actual queue.

A simple case is that of single-argument asynchronous messages (here, **int** messages); these generate a thin wrapper on top of the standard queue library:

```
class intQ {
  private Queue q;
  public intQ() {q = new Queue(); }
  public void add(int i) { q.Enqueue(i); }
  public int get() {return (int) q.Dequeue(); }
  public bool empty {get{return q.Count == 0;}}
}
```

The queue classes for messages with multiple arguments use queues of arrays of objects. (Clearly, our implementation would benefit greatly from the availability of generics in C♯ [17].)

Another important case of message-queue deals with empty (no argument) messages, or signals. It is implemented as a single message counter.

```
class voidQ {
  private int n;
  public voidQ() { n = 0; }
  public void add() { n++; }
  public void get() { n--; }
  public bool empty {get{ return n==0; }}
}
```

Finally, for synchronous methods, we need classes implementing queues of waiting threads. As with message queues, there is a uniform interface and a choice of several implementations. In the interface, *empty*, *add*, and *pop* are similar to those for messages, except that *pop* does not return a value—we know that this value is the current thread.

More specifically, *yield* and *wakeup* deal with synchronization code. Method *yield* is called to store the current thread in the queue and awaits for additional messages; it assumes the thread holds a lock on a polyphonic object, and releases that lock while waiting. Conversely, method *wakeup* is called to wake up a thread in the queue; it immediately returns and does not otherwise affect its own thread.

The code below implements synchronization using *monitors*, the low-level interface to object locks in C♯.

```
class threadQ {
  private Queue q;
  public threadQ() { q = new Queue(); }
  private void add() { q.Enqueue(Thread.CurrentThread); }
  private void pop() { q.Dequeue(); }
  public bool empty {get{ return (q.Count == 0); }}
  public void yield(object myCurrentLock) {
    add();
    Monitor.Exit(myCurrentLock);
    try {
      Thread.Sleep(Timeout.Infinite);
    }
    catch (ThreadInterruptedException) {}
    Monitor.Enter(myCurrentLock);
    pop();
  }
  public void wakeup() {((Thread) q.Peek()).Interrupt();}
}
```

(The specification of monitors guarantees that an interrupt on a non-sleeping thread does not happen until the thread actually does call *Thread.Sleep*, hence it *is* correct to release the lock before entering the **try catch** statement.)

As the thread awakens in the **catch** clause, it first reacquires the lock, which might block the thread again; we expect this case to be uncommon. The thread de-queued and discarded by *pop*() is always the current thread.

4.2.2 Generated Synchronization Code

Figure 1 shows a simple polyphonic class *Token* and its translation into ordinary C♯, making use of the auxiliary classes defined above. *Token* implements an n-token lock. It has a regular synchronous method, an asynchronous method, and a single chord that synchronizes the two.

```
class Token {
  public Token(int initial_tokens) {
    for (int i = 0; i < initial_tokens ; i++) release();
  }
  public void grab(int id) & public async release() {
    return id;
  }
}
```

```
class Token {
  private const int m_grab = 1 << 0;
  private const int m_release = 1 << 1;
  private threadQ grabQ = new threadQ();
  private voidQ releaseQ = new voidQ();

  private const int m_grab_release = m_grab | m_release;
  private BitMask s = new BitMask();
  private object mlock = grabQ;

  private void scan() {
    if (s.match(m_grab_release)) grabQ.wakeup();
  }
  public Token(int initial_tokens) {
    for (int i = 0; i < initial_tokens ; i++) release();
  }
  [OneWay]
  public void release() {
    lock(mlock) {
      releaseQ.add();
      if (!s.match(m_release)) {
        s.set(m_release);
        scan(); }}
  }
  public int grab(int id) {
    Monitor.Enter(mlock);
    if (!s.match(m_grab)) goto now;
  later:
    grabQ.yield(mlock); if (grabQ.empty) s.clear(m_grab);
  now:
    if (s.match(m_release)) {
      releaseQ.get(); if (releaseQ.empty) s.clear(m_release);
      scan();
      Monitor.Exit(mlock);
      {
        return id; // source code for the pattern
      }
    }else{
      s.set(m_grab); goto later; }}
}
```

Figure 1: The *Token* class and its translation

We now describe what is happening in the translations of the two methods:

4.2.3 Details for *release*

After taking the chord lock, we add the message to the queue and, unless they were already messages stored in *releaseQ*, we update the mask and scan for active chords.

In a larger class with patterns that do not involve *release*, the *scan*() statement could be usefully inlined and specialized: we only need to test patterns where **async** *release*() appears; besides, we know that the *m_release* bit is set.

4.2.4 The use of OneWay

The reader unfamiliar with C♯ may wonder why '[**OneWay**]' appears before the translation of the *release* () method. This is a C♯ attribute[6] which indicates to the .NET infrastructure that where appropriate (e.g. when calling between different machines) calls of *release* () should be genuinely non-blocking. The translation adds this attribute to all asynchronous methods.

4.2.5 Details for *grab*

After taking the chord lock, we first check whether there are already deferred *grab*s stored in *grabQ*; then, this call cannot proceed for now: we enqueue the current thread and will retry later.

Otherwise, we check whether there is (a least) a pending *release* message to complete the pattern *grab*(**int** *id*) & **async** *release*(); then, we can select this pattern for immediate execution, and otherwise we update the mask to record the presence of deferred *grab*s, enqueue the current thread, and will retry later. (In classes with multiple patterns for *grab*, we would perform a series of test for each potential pattern.)

Notice that it is always safe to retry, independently of the synchronization state.

Once a pattern is selected, we still have to update *releaseQ* and the mask. (Here, we don't have asynchronous parameters; more generally, we would read them from the queue and bind them to local variables.) At least in some cases, we must check whether there are still enough messages to awaken another thread; this is achieved by *scan*(). Finally, we release the lock and enter the block associated with the selected pattern; this code is executed in the same context as in the source program.

4.2.6 Why rescanning?

The reader may wonder why we systematically call *scan*() after selecting a chord for immediate execution (just before releasing the lock and executing the guarded block). In our simple example, this is unnecessary whenever we already know that this was the last *scan*() call or the last *release* () message. In general, however, this may be required to prevent deadlocks. Consider for instance the polyphonic class

```
class Foo {
  void m1() & async s() & async t() {...}
  void m2() & async s() {...}
  void m3() & async t() {...}
}
```

[6] Attributes are a standardized, declarative way of adding custom metadata to .NET programs. Code-manipulating tools and libraries, such as compilers, debuggers or the object serialization libraries can then use attribute information to vary their behaviour.

and the following global execution trace, with four threads:

Thread 1 calls *m1*() and blocks.

Thread 2: calls *m2*() and blocks.

Thread 0: calls *t*() then *s*(), awaking Thread 1

Thread 3: calls *m3*() and succeeds, consuming *t*().

Thread 1: retries *m1*() and blocks again.

With this scheduling, Thread 3 preempts Thread 1 and "steals" its message *t*(). Although Thread 1 blocks again, the remaining message *s*() suffices to run Thread 2. But if neither Thread 3 nor Thread 1 awakes Thread 2, we have a race condition leading to a deadlock.

Accordingly, in our implementation, the synchronization code in Thread 3 performs an additional *scan*() that awakes Thread 2 in such unfortunate cases.

In many special cases, the final *scan*() could safely be omitted; however, this would unnecessarily complicate the translation.

4.2.7 Deadlock Freedom

Next, we sketch a proof that our translation does not introduce deadlocks. (Of course, calls involving a chord that is never fired may be deadlocked, and our translation must implement those deadlocks.)

We say that an object is *active* when there are enough calls in the queues to trigger one of its patterns; assuming a fair scheduling of running processes, we show that active states are transient. We prove the invariant: *when an object is active, at least one thread on top of a queue is scheduled for execution and can succeed.*

- After *scan*(), the invariant always holds.
- An object becomes active when an asynchronous message is received, and this always triggers a scan.
- A thread whose polyphonic call succeeds (and thus consumes asynchronous messages) also triggers a scan.

When the algorithm awakes a thread, it is guaranteed that this thread may succeed if immediately scheduled, but not that it will necessarily succeed.

4.2.8 Fully Asynchronous Patterns

To complete the description of our implementation, we explain the compilation of fully asychronous chords. When such chords are fired, there is no thread at hand to execute their body, so a new thread must be created.

To illustrate this case, assume the class *Token* also contains the asynchronous method declaration

```
public async live(string s,int id) {
  grab(id); release();
  Console.WriteLine(s);
}
```

The generated code is large but straightforward:

```
  private class liveArgs {
    public string s; public int id;
    public liveArgs(string s, int id) {
      this.s = s; this.id = id;
    }
  }
  private void liveBody(object o) {
    liveArgs  a = (liveArgs)o;
    string s = a.s; int id = a.id;
    {
      grab(id); release();
      Console.WriteLine(s);
    }
  }
  [OneWay]
  public void live(string s,int id) {
    liveArgs  a = new liveArgs(s,id);
    WaitCallback c = new WaitCallback(liveBody);
    ThreadPool.QueueUserWorkItem(c,a);
  }
}
```

We use an auxiliary class *liveArgs* to pass the parameters to the new thread, and a delegate to the host object's *liveBody* method to resume execution within the same object context.

More generally, for a chord containing several asynchronous methods, the code in the *live* method above would occur instead of *mQ.wakeup*() to fire the pattern in method *scan*().

5 Current Status and Future Work

We have two prototype implementations of Polyphonic C♯. The first is a modifed version of the 'official' C♯ compiler, which is written in C++, whilst the second is a simpler source-to-source translator written in ML. The latter has proven invaluable in explaining the language to others and is also considerably more straighforward to modify and maintain, though it does not cope with the full language. As our initial experiences using Polyphonic C♯ have been positive, we are building a more robust, full-featured and maintainable implementation built on an 'experimentation-friendly' C♯-in-C♯ compiler being produced by another group within Microsoft Research.

We have written a number of non-trivial samples in Polyphonic C♯, including some web combinators along the lines of [5], an animated version of the dining philosphers and a distributed stock-dealing simulation built on .NET's remoting infrastructure.[7]

Amongst the other areas for further work on Polyphonic C♯ which we think are particularly interesting are:

Concurrency Types As suggested in our examples, it is relatively easy to state and verify invariants in polyphonic classes, often from the shape of the chords and the visibility of their methods.

Several type systems and other static analyses have been developed in similar setting to automate the process, and check (or even infer) at compile time some behavioral properties such as

1. There is one, or at most one, pending message for this asynchronous method, or for this set of methods.
2. Calls to this method are always eventually processed (partial deadlock-freedom).

The potential benefits are obvious: the compiler can catch more programming errors, and otherwise produce more efficient code. While these tools are still rather complex, this is a very active area of research in concurrency. (Needless to say, it would be much more difficult to check those properties on a code that directly uses threads and locks instead of chords.)

Timeouts and Priorities In terms of expressiveness, it is tempting to supplement the syntax for chords with some declarative support for priorities or timeouts and, more generally, to provide a finer control over dynamic scheduling. We have a plausible-looking design for a timeout mechanism which we plan to implement and evaluate soon.

Optimizations There are many opportunities for optimizing the simple-minded implementation described here. Some of these require proper static analysis, whereas others could usefully be implemented on the basis of more naive compile-time checks:

- Lower-level concurrency control. There are situations when we could safely 'fuse' successive critical sections which are protected by the same lock, for example when a bounded series of asynchronous messages are sent to the same object, or when a chord body sends messages to **this**. Purely asynchronous chords which only perform very brief terminating computations (such as sending other messages) can also be compiled to run in the invoking thread, rather than a new one.
- Queue optimization. 'Affine' methods, for which it can be determined that there can be at most one pending call on a particular object, may be compiled without queues.

6 Conclusions

Asynchronous concurrent programming is becoming more important and widespread but is still extremely hard. We have designed and implemented a Join-based extension of C♯ which is simple, expressive, and can be efficiently implemented. In our experience, writing correct concurrent programs is considerably less difficult in Polyphonic C♯ than in ordinary C♯ (though we would certainly not go so far as to claim that it is *easy*...).

The integration of the Join-based constructs with objects and the existing platform support for concurrency is not entirely straightforward – our implementation is slightly constrained by the threads and locks model and some uses of existing libraries and frameworks require a little 'impedence matching'. Nevertheless, the new constructs seem to work well in practice.

Acknowledgements Thanks to Mark Shinwell, who did the initial implementation work on the Polyphonic C♯ compiler during an internship at Microsoft Research.

[7] Remoting provides object-oriented RPC over TCP (binary) or HTTP (SOAP).

References

[1] Gul Agha, Peter Wegner, and Akinori Yonezawa. *Research Directions in Concurrent Object-Oriented Programming*. MIT Press, 1993.

[2] Pierre America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1(4):366–411, 1989.

[3] A. D. Birrell. An introduction to programming with threads. Research Report 35, DEC SRC, January 1989.

[4] A. D. Birrell, J. V. Guttag, J. J. Horning, and R. Levin. Synchronization primitives for a multiprocessor: A formal specification. Research Report 20, DEC SRC, August 1987.

[5] Luca Cardelli and Rowan Davies. Service combinators for web computing. *Software Engineering*, 25(3):309–316, 1999.

[6] S. Conchon and F. Le Fessant. Jocaml: Mobile agents for Objective-Caml. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, pages 22–29. IEEE Computer Society, October 1999. Software and documentation available from `http://pauillac.inria.fr/jocaml`.

[7] David L. Detlefs, K. Rustan, M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, DEC SRC, December 1998.

[8] C. Fournet and G. Gonthier. The join calculus: a language for distributed mobile programming. In *Proceedings of the Applied Semantics Summer School (APPSEM), Caminha, September 2000*. To appear. Draft available from `http://research.microsoft.com/~fournet`.

[9] C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proc. POPL'96*, pages 372–385. ACM, January 1996.

[10] C. Fournet, C. Laneve, L. Maranget, and D. Rémy. Inheritance in the join-calculus (extended abstract). In *FST TCS 2000: Foundations of Software Technology and Theoretical Computer Science*, volume 1974 of *LNCS*, pages 397–408, New Delhi, India, December 2000. Springer-Verlag. Full version available from `http://research.microsoft.com/~fournet`.

[11] J. Gosling, B. Joy, and G. Steele. Threads and locks. In *The Java Language Specification*, chapter 17. Addison Wesley, 1996.

[12] Y. Gurevich, W. Schulte, and C. Wallace. Investigating java concurrency using abstract state machines. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines: Theory and Applications*, volume 1912 of *Lecture Notes in Computer Science*, pages 151–176. Springer, 2000.

[13] A. Hejlsberg and S. Wiltamuth. *C♯ Language Specification*. Microsoft Corporation, 2000.

[14] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[15] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[16] S. Kamin, editor. *Proceedings of the First ACM-SIGPLAN Workshop on Domain-Specific Languages*, Paris, France, January 1997.

[17] A. Kennedy and D. Syme. Design and implementation of generics for the .NET Common Language Runtime. In *Proceedings of the ACM SIGPLAN Conference of Programming Language Design and Implementation*. ACM, 2001.

[18] James R. Larus and Michael Parkes. Using cohort scheduling to enhance server performance. Technical Report MSR-TR-2001-39, Microsoft Research, March 2001.

[19] F. Le Fessant and L. Maranget. Compiling join-patterns. In Uwe Nestmann and Benjamin C. Pierce, editors, *HLCL '98: High-Level Concurrent Languages*, volume 16(3) of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, September 1998.

[20] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, second edition edition, 1999.

[21] INMOS Limited. *Occam Programming Manual*. Prentice-Hall Int., 1984.

[22] Satoshi Matsuoka and Akinori Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In AghaWegnerYonezawa [1], chapter 4, pages 107–150.

[23] M. Odersky. Functional nets. In *Proc. European Symposium on Programming*, volume 1782 of *LNCS*, pages 1–25. Springer Verlag, 2000.

[24] Michael Philippsen. Imperative concurrent object-oriented languages: An annotated bibliography. Technical Report TR-95-049, International Computer Science Institute, Berkeley, CA, 1995.

[25] J. C. Ramming, editor. *Proceedings of the First USENIX Conference on Domain-Specific Languages*, Santa Barbara, California, October 1997.