The `Joins` Concurrency Library

Claudio Russo

Microsoft Research Ltd, 7JJ Thomson Ave, Cambridge, United Kingdom
crusso@microsoft.com

Abstract. Cω extended C$^\#$ 1.x with a simple, declarative and powerful model of concurrency - join patterns - applicable both to multithreaded applications and to the orchestration of asynchronous, event-based distributed applications. With Generics available in C$^\#$ 2.0, we can now provide join patterns as a library rather than a language feature. The `Joins` library extends its clients with an embedded, type-safe and mostly declarative language for expressing synchronization patterns. The library has some advantages over Cω: it is language neutral, supporting other languages like Visual Basic; its join patterns are more dynamic, allowing solutions difficult to express with Cω; its code is easy to modify, fostering experimentation. Although presenting fewer optimization opportunities, the implementation is efficient and its interface makes it trivial to translate Cω programs to C$^\#$. We describe the interface and implementation of Joins which (ab)uses almost every feature of Generics.

1 Introduction

Cω [1] promised C$^\#$ 1.x users a more pleasant world of concurrent programming. Cω presents a simple, declarative and powerful model of concurrency - *join patterns* - applicable both to multithreaded applications and to the orchestration of asynchronous, event-based distributed applications. Using Generics in C$^\#$ 2.0 (and the .NET runtime in general), we can now provide join patterns as a .NET library – called `Joins` – rather than a language extension. Encoding language features in a library has some obvious drawbacks, restricting the scope for both optimization and static checking – but it also has some advantages. The `Joins` library is language neutral; it can be used by C$^\#$ but also by Visual Basic and other .NET languages. A library can be more dynamic: the `Joins` library already supports solutions that are more difficult to express with the declarative join patterns of Cω (Section 3.1). A library is easier to modify than a compiler, promoting experimentation. The `Joins` implementation is reasonably efficient and takes advantage of the same basic optimizations performed by the Cω compiler. Its interface makes it particularly easy to translate Cω programs to C$^\#$, but it can also be used to write concurrent code from scratch.

Section 2 presents join patterns as found in Cω. Section 3 introduces the `Joins` library by example, showing how to re-express the Cω programs of Section 2 as C$^\#$ 2.0 code that references the library. Section 4 provides a concise, yet precise, description of the `Joins` library as it appears to the user. Section 5 gives

an overview of the implementation which exercises most features of Generics. Section 6 concludes, discussing related work. The `Joins` download and tutorial [2] presents many more examples including encodings of active objects or actors, bounded buffers, reader/writer locks, futures, dining philosophers, a lift controller and simple, distributed applications using web services and Remoting.

2 Background: Cω's concurrency constructs

Cω extends the C# 1.2 programming language with new asynchronous concurrency abstractions. The new constructs are a mild syntactic variant of those previously described under the name 'Polyphonic C#' [3]. Similar extensions to Java were independently proposed by von Itzstein and Kearney [4].

In Cω, methods can be defined as either *synchronous* or *asynchronous*. When a synchronous method is called, the caller is blocked until the method returns, as is normal in C#. However, when an asynchronous method is called, there is no result and the caller proceeds immediately without being blocked. Thus from the caller's point of view, an asynchronous method is like a `void` one, but with the useful extra guarantee of returning immediately. We often refer to asynchronous methods as *messages*, as they are one-way communications.

By themselves, asynchronous method declarations are not particularly novel: the innovation of Cω is the way method bodies are defined. In most languages, including C#, methods in the signature of a class are in bijective correspondence with the code of their implementations. In Cω, however, a body may be associated with a *set* of synchronous and/or asynchronous methods, including at most one synchronous method. Such definitions are called *chords* and a particular method may appear in the header of several chords. The body of a chord can only execute once *all* the methods in its header have been called. Calling a chorded method may thus enable zero, one or more chords:

- If no chord is enabled then the method invocation is queued up. If the method is asynchronous, then this simply involves adding the arguments (the contents of the message) to a queue. If the method is synchronous, then the calling thread is blocked.
- If there is a single enabled chord, then the arguments of the calls involved in the match are de-queued, and any blocked thread involved in the match is awakened to run the chord's body in that thread. The body of a chord involving only asynchronous methods runs in a new thread.
- If several chords are enabled, an unspecified one is selected to run.
- If multiple calls to one method are queued up, which call will be de-queued by a match is left unspecified.

Here is the simplest interesting example of a Cω class:

```
public class Buffer {
   public async Put(string s);
   public string Get() & Put(string s) { return s; }
}
```

This class contains two methods: a synchronous one, `Get()`, which takes no arguments and returns a string, and an asynchronous one, `Put(s)`, which takes a string argument and (like all asynchronous methods) returns no result. The class definition contains two things: a declaration (with no body) for the asynchronous method, and a chord. The chord declares the synchronous method and defines a body (the return statement) which can run when *both* the `Get()` and `Put(s)` methods have been called.

Now assume that producer and consumer threads wish to communicate via an instance `b` of the class `Buffer`. Producers make calls to `b.Put(s)`, which, since the method is asynchronous, never block. Consumers make calls to `b.Get()`, which, since the method is synchronous, will block until or unless there is a matching call to `Put(s)`. Once `b` has received both a `Put(s)` and a `Get()`, the body runs and the actual argument to `Put(s)` is returned as the result of the call to `Get()`. Multiple calls to `Get()` may be pending before a `Put(s)` is received to reawaken one of them, and multiple calls to `Put(s)` may be made before their arguments are consumed by subsequent `Get()`s. Note that:

1. The body of the chord runs in the (reawakened) thread corresponding to the matched call to `Get()`. Hence no new threads are spawned in this example.
2. The code which is generated by the class definition above is completely thread safe. The compiler generates the necessary locking. Furthermore, the locking is fine-grained and brief - chorded methods do not lock the whole object and are not executed with "monitor semantics".
3. The return value of a chord is returned to its synchronous method, of which there can be at most one.

In general, the definition of a synchronous method in Cω consists of more than one chord, each of which defines a body that can run when the method has been called *and* a particular set of asynchronous messages are present. For example, we could modify the example above to allow `Get()` to synchronize with calls to either of two different `Put1(s)` and `Put2(n)` methods:

```
public class BufferTwo {
  public async Put1(string s); public async Put2(int n);
  public string Get() & Put1(string s) { return s;}
                      & Put2(int n) { return n;} // ie. n.ToString()
}
```

Now we have two asynchronous methods and a synchronous method which can synchronize with either one, with a different body in each case.

A chord may involve more than one message; this synchronous chord waits for messages on both `Put1` and `Put2`:

```
  public string Both() & Put1(string s) & Put2(int n) { return s + n;}
```

In Cω, a purely asynchronous chord is written as a class member, like this:

```
  when Put1(string s) & Put2(int n) { Console.WriteLine(s + n);}
```

This chord spawns a new thread when messages arrive on `Put1` and `Put2`.

The previous `Buffer` class is unbounded: any number of calls to `Put(s)` could be queued up before matching a `Get()`. We now define a variant in which only a single data value may be held in the buffer at any one time:

```
public class OnePlaceBuffer {
   private async Empty();
   private async Contains(string s);
   public void Put(string s) & Empty() { Contains(s); }
   public string Get() & Contains(string s) { Empty(); return s;}
   public OnePlaceBuffer() { Empty(); }
}
```

The public interface of `OnePlaceBuffer` is similar to that of `Buffer`, but the `Put(s)` method is now synchronous and will block if there is already an unconsumed value in the buffer.

The implementation of `OnePlaceBuffer` makes use of two *private* asynchronous messages: `Empty()` and `Contains(s)`. These are used to carry the state of the buffer and illustrate a very common programming pattern in Cω. The class is best understood by reading its code declaratively:

- When a new buffer is created, it is initially `Empty()`.
- If you call `Put(s)` on an `Empty()` buffer then it subsequently `Contains(s)` and the call to `Put(s)` returns.
- If you call `Get()` on a buffer which `Contains(s)` then the buffer is subsequently `Empty()` and `s` is returned to the caller of `Get()`.
- Implicitly, in all other cases, calls to `Put(s)` and `Get()` block.

The constructor establishes and the chords maintain the invariant that there is always exactly one `Empty()` or `Contains(s)` message pending on the buffer. The chords can easily be read as the specification of a finite state machine.

3 The `Joins` Library

In Cω, classes that declare (a)synchronous methods joined in chords implicitly declare a set of communication channels. An asynchronous method has a backing queue of pending method calls. A synchronous method has a backing queue of waiting threads. The state of the queues is protected by a hidden lock. Invoking an (a)synchronous method executes some specialized scheduling code that decides, given the current queues and the declared chords, which, if any, chord gets to fire, either on the current or any waiting thread. Thus each object (or, for purely static methods, class) includes its own scheduling logic. Instead of relying on a central scheduling thread, threads that invoke chorded methods each spend a little time helping to schedule each other. To optimize the detection of enabled chords, the implementation maintains some additional state: a bit vector representing the set of non-empty queues. Pattern matching is compiled to subset tests against this state, implemented using one bitmask operation per chord.

In the `Joins` library, the scheduling logic that would be compiled into the corresponding Cω class receives a separate, first-class representation as an object of the special `Join` class. The `Join` class provides a mostly declarative, type-safe mechanism for defining thread-safe synchronous and asynchronous communication channels and patterns. Instead of (a)synchronous methods, as in Cω, the communication channels are special delegate values (first-class methods) obtained from a common `Join` object. Communication and/or synchronization takes place by invoking these delegates, passing arguments and optionally waiting for return values. The allowable communication patterns as well as their effects are defined using *join patterns*: bodies of code whose execution is guarded by linear combinations of channels. The body, or *continuation*, of a join pattern is provided by the user as a (typically anonymous) delegate that can manipulate external resources protected by the `Join` object.

Using the `Joins` library, we can implement the Cω `Buffer` in C$^\#$ as follows:

```
using Microsoft.Research.Joins;
public class Buffer {
  // Declare the (a)synchronous channels
  public readonly Asynchronous.Channel<string> Put;
  public readonly Synchronous<string>.Channel Get;
  public Buffer() {
    // Create a Join object
    Join join = Join.Create();
    // Use it to initialize the channels
    join.Initialize(out Put); join.Initialize(out Get);
    // Finally, declare the patterns(s)
    join.When(Get).And(Put).Do(delegate(string s) { return s;});
  }}
```

The code declares a buffer class with two fields of special delegate types. The `Put` field contains an asynchronous channel that, when invoked, returns `void` (immediately) and *takes* one `string` argument. The `Get` field contains a synchronous channel that, when invoked, *returns* a `string` but takes no argument. Both fields are initially `null`. The constructor allocates a new `Join` object, `join`, using the factory method `Join.Create`. The `join` object is a private scheduler for the buffer. The constructor then calls `Initialize` on `join`, passing the locations of each of the channels: this assigns two new delegate values into the fields, each obtained from and owned by `join`. Finally we declare the Cω chord by constructing a pattern on the `join` object, passing the synchronous channel `Get` to `When` and `And`ing it with the asynchronous channel `Put`. The pattern is completed by invoking `Do`, passing the continuation for this pattern, expressed here as an anonymous delegate. The continuation expects exactly one argument (the argument to `Put`); the continuation's return value is returned to the caller of `Get`. Notice that the bodies of the continuation and Cω chord are identical.

If we ignore the boilerplate calls to `Initialize` then what remains retains the declarative flavour of the original Cω code. Moreover, client code of the Cω and C$^\#$ buffers is syntactically identical. Given a buffer `b`, clients invoke

`b.Put(s);` to send a string `s` and `b.Get()` to receive one. Of course, these calls are compiled slightly differently, just invoking a method in the Cω client, but reading a field and then invoking its delegate value in the C$^\#$ client.

A synchronous method with several chords translates to several patterns constructed on the same initial channel. In general, calls to `And` may be iterated, and a continuation may bind zero or more parameters and return zero or one values, depending on the pattern. An asynchronous chord translates to a pattern with an initial asynchronous channel whose continuation returns `void`.

Here is Cω's `OnePlaceBuffer`, made generic in C$^\#$ for good measure:

```
public class OnePlaceBuffer<S> {
  private readonly Asynchronous.Channel Empty;
  private readonly Asynchronous.Channel<S> Contains;
  public readonly Synchronous.Channel<S> Put;
  public readonly Synchronous<S>.Channel Get;
  public OnePlaceBuffer() {
    Join j = Join.Create();
    j.Initialize(out Empty); j.Initialize(out Contains);
    j.Initialize(out Put); j.Initialize(out Get);
    j.When(Put).And(Empty).Do(delegate(S s) { Contains(s);});
    j.When(Get).And(Contains).Do(delegate(S s) { Empty(); return s;});
    Empty();
  }}
```

`Empty` and `Put` introduce two more channel types. An `Asynchronous.Channel` delegate takes zero arguments and returns void. As in Cω, nullary channels use a more efficient counter instead of a queue of argument values to record pending invocations. A `Synchronous.Channel<S>` delegate returns `void` and takes one argument of type `S`. To protect the buffer's invariant, we translate the private Cω `Empty` and `Contains` messages to private fields, accessible from the continuations but not externally. The constructor establishes the invariant by calling `Empty()`, after initializing the channels and constructing the patterns.

3.1 Beyond Cω: Dynamic Joins

What if we need to declare, and synchronize, a dynamic set of channels? A Cω class can only declare a static set of channels and chords so a dynamic set has to be encoded by resorting to multiplexing. Although possible, this is inconvenient. Inspired by a similar feature in the CCR [5], the `Joins` library lets you initialize, and join *arrays* of asynchronous channels. Since the size of an array is determined at runtime, this supports *dynamic* synchronization patterns.

For example, the `JoinMany<`R`>` class below declares and supports waiting on n channels of type R, which is awkward to express in Cω. The class declares an array, `Responses`, of response channels, each carrying a value of type R. An object o `= new JoinMany<`R`>`(n) requires $n+1$ channels: n asynchronous response channels, o`.Responses[i]` $(0 \leq i < n)$, and one synchronous channel, o`.Wait`. The constructor `Create`s a `Join` object supporting $n+1$ channels; it

then `Initialize`s the response channels field with an array of n distinct channels and declares a pattern that waits on all the channels in this array. The continuation of the pattern receives all of the responses as a separate array (also of size n) of correlated values of type R. The consumer calls o`.Wait`(), blocking until/unless all responses have arrived; producer i just posts her response r on o`.Response(`i`)(`r`)`, asynchronously. Here, we have taken the precaution of hiding the array in a private field to prevent external updates – we could avoid this if C# supported immutable arrays or we bothered to roll our own.

```
public class JoinMany<R> {
  private readonly Asynchronous.Channel<R>[] Responses;
  public readonly Synchronous<R[]>.Channel Wait;
  public Asynchronous.Channel<R> Response(int i) { return Responses[i]; }
  public JoinMany(int n) {
    Join j = Join.Create(n + 1);
    j.Initialize(out Responses, n); j.Initialize(out Wait);
    j.When(Wait).And(Responses).Do(delegate(R[] r) { return r; });
  }}
```

4 `Joins` Library Reference

Users of `Joins` reference the assembly `Microsoft.Research.Joins.dll` and import the namespace `Microsoft.Research.Joins`.

A new `Join` instance j is allocated by calling an overload of factory method `Join.Create`([*size*]). The optional integer *size* bounds the number of channels supported by j and defaults to 32; it also sets the constant property j`.Size`.

A `Join` object notionally owns a set of asynchronous and synchronous *channels*, each obtained by calling an overload of method `Initialize`, passing the location of a *channel* or array of *channels* using an `out` argument:

$$j\texttt{.Initialize(out }\textit{channel}\texttt{)};\ \texttt{or}\ j\texttt{.Initialize(out }\textit{channels}, \textit{length}\texttt{)};$$

The second form assigns to location *channels* an *array* of *length* distinct, asynchronous channels. It is possible to initialize the same location twice.

Channels are instances of the following delegate types, summarized by a simple grammar of type expressions:

$$(\texttt{Asynchronous} \mid \texttt{Synchronous}[\langle R\rangle])\texttt{.Channel}[\langle A\rangle]$$

The outer class of a channel, `Asynchronous`, `Synchronous` or `Synchronous<`R`>`, should be read as a modifier that specifies its blocking behaviour and optional return type R. Type A, if present, determines the channel's optional argument type. The six channel flavours support zero or one arguments of type A and zero or one results of type R. Multiple arguments or results must be passed in tuples, either using the provided generic `Pair<`A, B`>` struct or by other means.

Apart from its channels, a `Join` object notionally owns a set of *join patterns*. A join pattern is constructed by invoking an overload of the instance method

`When` followed by zero or more invocations of instance method `And` (or `AndPair`), followed by a final invocation of instance method `Do`. A constructed join pattern typically takes the form:

$$j\texttt{.When}(a_1)\texttt{.And}(a_2)\cdots\texttt{.And}(a_n)\texttt{.Do}(d);$$

Alternatively, using an anonymous delegate for d:

$$j\texttt{.When}(a_1)\texttt{.And}(a_2)\cdots\texttt{.And}(a_n)\texttt{.Do(delegate}(P_1\ p_1,\ldots,P_m\ p_m)\{\ldots\});$$

Argument a_1 of $\texttt{When}(a_1)$ may be a synchronous or asynchronous channel or an array of asynchronous channels. Each subsequent argument a_i to $\texttt{And}(a_i)$ (for $i > 1$) must be an asynchronous channel or an array of asynchronous channels; it cannot be a synchronous channel. The argument d to $\texttt{Do}(d)$ is a *continuation* delegate that defines the body of the pattern. Although its precise type varies with the pattern, the continuation always has a delegate type of the form:

$$\texttt{delegate [void | }R\texttt{] Continuation}(P_1\ p_1,\ldots,P_m\ p_m);$$

The precise type of the continuation d, including its arity or number of arguments m, is determined by the sequence of channels guarding it. If the first argument a_1 in the pattern is a synchronous channel with return type R, then the continuation's return type is R; otherwise the return type is `void`.

The continuation receives the arguments of the joined channel invocations as delegate parameters $P_1\ p_1,\ldots,P_m\ p_m$, for $m \leq n$. The presence and types of any additional parameters $P_1\ p_1,\ldots,P_m\ p_m$ varies according to the type of each argument a_i joined with invocation $\texttt{When}(a_i)/\texttt{And}(a_i)$ (for $1 \leq i \leq n$):

- If a_i is of type `Channel` or `Channel[]`, then $\texttt{When}(a_i)/\texttt{And}(a_i)$ adds no parameter to delegate d.
- If a_i is of type `Channel<`P`>` or `Channel<`P`>[]` then $\texttt{When}(a_i)/\texttt{And}(a_i)$ adds one parameter p_j of type $P_j = P$ or $Pj = P[]$ (respectively) to delegate d.

Parameters are added to d from left to right, in increasing order of i. A continuation can receive at most $m \leq max$ parameters ($max = 8$ in the current implementation). If necessary, it is possible to join more than max generic channels by calling method $\texttt{AndPair}(a_i)$ instead of $\texttt{And}(a_i)$. $\texttt{AndPair}(a_i)$ modifies the last argument of the new continuation to be a pair consisting of the last argument of the previous continuation and the new argument contributed by ai.

Readonly property j`.Count` is the current number of channels initialized on j; it is bounded by j`.Size`. Any invocation of j`.Initialize` that would cause j`.Count` to exceed j`.Size` throws `JoinException`. Join patterns must be well-formed, both individually and collectively. Executing $\texttt{Do}(d)$ to complete a join pattern will throw `JoinException` if d is `null`, the pattern repeats an asynchronous channel (i.e. is non-linear), an (a)synchronous channel is `null` or *foreign* to this pattern's `Join` instance, the join pattern is *redundant*, or the join pattern is *empty*. A channel is foreign to a `Join` instance j if it was not allocated by some call to j`.Initialize`. A pattern is redundant when the set of channels joined by the pattern subsets or supersets the channels joined by another pattern on this `Join` instance. A pattern is empty when its set of channels is empty.

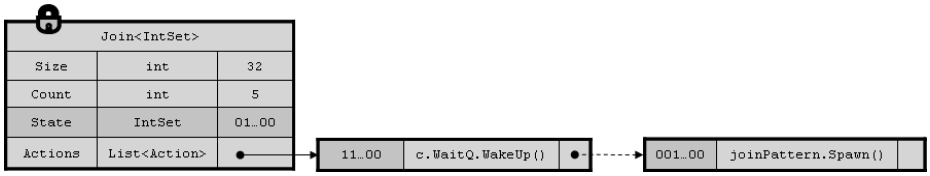
5 Implementation

The implementation avoids using Reflection and only uses checked casts to extract the underlying queue from a channel when constructing a pattern. These casts could have been avoided by defining the channel delegates to contain queue fields (possible in bytecode, but not C#), or by representing channels as *classes*. We preferred to retain the convenient delegate invocation syntax for sending messages and to provide a pure C# implementation. To be useful in practice, we provide 6 flavours of channel rather than two basic ones (`Asynchronous.Channel<A>` and `Synchronous<R>.Channel<A>`) because passing or returning ML-like `unit` values is just unnatural in C# and VB. We favour n-ary continuations, despite the (soft) limit on n, because uniform currying is awkward in C# and unsupported in VB; similarly, without pattern matching, using uniformly nested pairs to bind continuation arguments requires unwieldy projections. Compare the first `void`-returning, 3-argument continuation with its uglier, but more "uniform" alternatives:

```
1. delegate(int i,bool b,float f){ Console.Write("{0},{1},{2}",i,b,f);}
2. delegate(int i){return delegate(bool b){return delegate(float f){
    Console.Write("{0},{1},{2}", i, b, f); return new Unit();};};}
3. delegate(Pair<Pair<int, bool>,float> p){
    Console.Write("{0},{1},{2}", p.Fst.Fst, p.Fst.Snd, p.Snd); }
```

5.1 Join and Channel Object Representations

The `Join` class is abstract. Each `Join` object j has runtime type `Join<IntSet>`, a specific instantiation of a private, overloaded generic class `Join<S>` that subclasses `Join`. `IntSet` is a *struct* type that implements a set of j`.Size`-bounded integers as a packed sequence of bits. A `Join<IntSet>` object looks like this:



It contains the following fields:

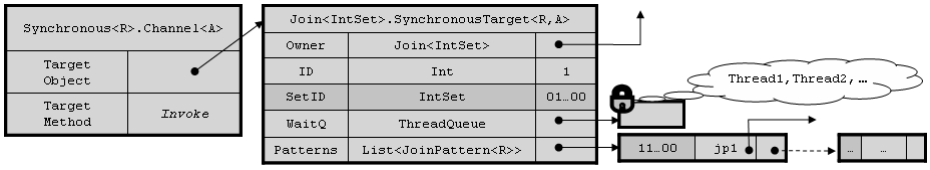
`Size`: an immutable bound on the number of channels that may be owned.
`Count`: the mutable, current number of channels owned by the instance and the `ID` of the next channel, incremented by calls to `Initialize`.
`State`: a mutable `IntSet` with a capacity of at least `Size` elements. `State` encodes the current set of non-empty channels as a set of channel `ID`s. Since `IntSet` is a struct, `State` is inlined in the object, not stored on the heap.
`Actions`: a mutable, `IntSet`-indexed list of pattern match *actions*: each action either wakes up one thread waiting on a synchronous channel's `WaitQ` or spawns the continuation of an asynchronous pattern on a new thread.

The regular object lock on a `Join` instance protects both its own state and the states of its channels. `Actions` is extended (under the `Join`'s lock) whenever a legal pattern is completed by calling a `Do` method. Registering a pattern pre-computes its `IntSet` for faster matching and early error detection.

Channels are delegates and thus contain a target object and a target method, comparable to the environment and code pointer of a closure in functional languages. All channel target objects contain the following immutable fields.

`Owner`: a reference to the `Join<IntSet>` instance that initialized the channel.
`ID`: an identifier for the channel, unique for the channels of `Owner`.
`SetID`: a pre-computed `IntSet` corresponding to the singleton set {`ID`}.

A `Synchronous<R>` channel, for example, looks like this:



Its target object additionally contains these fields:

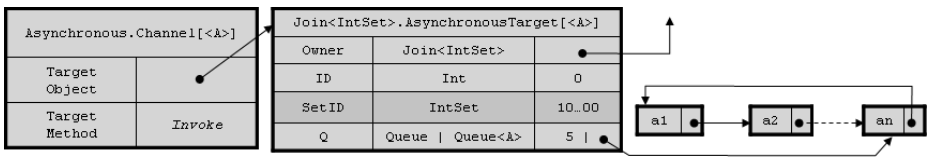
`WaitQ`: a notional queue of waiting threads, itself implemented using the implicit waitset of a privately allocated lock as in [3]. The `ThreadQ.WakeUp` method efficiently targets at most one waiting thread, avoiding `Monitor.PulseAll()`.
`Patterns`: an `IntSet`-indexed list of all `R`-returning patterns containing `ID`.

When invoked, the channel's target method acquires the `Owner`'s lock , scans `Patterns` for matches with the `Owner`'s `State` and either:

If there is no matching pattern: enqueues its thread, updates `State`, releases the `Owner` lock and blocks awaiting notification on the `WaitQ` lock.
If there is some matching pattern: dequeues the asynchronous channels involved in the pattern, updating `State`, scans for any enabled actions[1], releases the `Owner`'s lock and returns the value of invoking the pattern's continuation with the dequeued values in the current thread. Since the channel and continuation both return a value of type `R`, this involves no casting.

When it wakes up, the waiting thread re-acquires the `Owner`'s lock, and re-attempts to find a match amongst its patterns. If it fails, because some intervening thread has consumed some channel values available when the thread was awoken, the thread blocks, resuming its wait for a match.

The target object of a `Asynchronous` channel contains just one additional field, a queue `Q` of pending calls, so a `Channel[<A>]` looks like this:



[1] the additional scan is used to avoid deadlock – see [3] for a discussion.

The representation of `Q` depends on the channel's arity. A `Channel<A>` contains a proper FIF0 queue of type `Queue<A>`, implemented as a circular list of `A`-values with constant time access to both ends of the queue. A nullary and thus data-less `Channel` contains an optimized `Queue` struct, implemented in constant space by just recording the current *count* of notional queue entries.

When invoked, the channel's target method acquires its `Owner`'s lock and enqueues its argument or bumps its counter; if `Q` was empty, it updates `Owner`'s `State` and performs some action enabled by its new `State` (if any); finally, the method releases its `Owner`'s lock and returns. Assuming no malicious third party has grabbed the `Owner`'s lock, which is easily prevented by keeping all `Join` objects private, executing the action and the channel invocation is guaranteed to return since the lock is only held briefly by other channels.

5.2 Exploiting Generics

The `Joins` library makes extensive use of C# language features to present an API that we hope is relatively simple to use: a user only has to know a handful of identifiers and understand a simple grammar of channel types and join patterns. We rely on overloading and type argument inference to implicitly resolve method calls, that, were they explicit, would obscure the user's intentions.

The various channel flavours of Section 3 are implemented as (generic) delegate types, nested within (generic) static classes:

```
static class Asynchronous   { delegate void Channel();
                              delegate void Channel<A>(A a);}
static class Synchronous    { delegate void Channel();
                              delegate void Channel<A>(A a);}
static class Synchronous<R> { delegate R Channel();
                              delegate R Channel<A>(A a);}
```

Using both nesting and generic arity to overload the `Channel` identifier makes it easy for a user to independently change the blocking behaviour, argument and return type a channel.

The `Join` class provides essentially three methods: `Create`, `Initialize` and `When` and two integer properties `Count` and `Size` which are rarely needed:

```
abstract class Join {
  static Join Create([int size]);
  void Initialize[<A>](out Asynchronous.Channel[<A>] c);
  void Initialize[<A>](out Synchronous.Channel[<A>] c);
  void Initialize<R[, A]>(out Synchronous<R>.Channel[<A>] c);
  void Initialize[<A>](out Asynchronous.Channel[<A>][] cs, int length);
  JoinPattern.OpenPattern[<P>] When[<P>](Asynchronous.Channel[<P>] c);
  JoinPattern.OpenPattern[<A>] When[<A>](Synchronous.Channel[<A>] c);
  JoinPattern<R>.OpenPattern[<A>] When<R[,A]>(
    Synchronous<R>.Channel[<A>] c);
  JoinPattern.OpenPattern[<P[]>] When[<P>](
    Asynchronous.Channel[<P>][] cs);
  int Count { get; } int Size { get; }}
```

`Create(int size)` is a factory method that, internally, uses polymorphic recursion to construct, at runtime, an `IntSet` struct with a capacity of `size` (or more) elements. The library defines primitive 32- and 64-element sets, `IntSet32` and `IntSet64`, represented as one field structs of unsigned integers or longs. Each implements a simple interface `IIntSet<S>` providing imperative operations on the integer set type `S`: i.e. `IntSet32` implements `IIntSet<IntSet32>`, `IntSet64` implements `IIntSet<IntSet64>`. A generic struct `PairSet<S>` with type parameter constraint `where S:IIntSet<S>` is used to construct a double-capacity set from a smaller set representation. Notice that `PairSet<S>` uses a recursive type constraint (a.k.a F-bounded polymorphism) to parameterize over a representation `S` supporting a set of operations on `S`. The concrete, generic class `Join<S>` also declares this constraint on `S` so it can access set operations to manipulate its otherwise parametric `State` field. In C#, calls to an interface method on a struct actually pass the *this* pointer by reference, not value, and can therefore mutate the original value. We exploit this feature, updating `State` fields in-place.

The `Initialize` method assigns the location of a channel or array of channels with a new (set of) channel(s) allocated and owned by `this Join` instance. The method has eight overloads (summarized above), some generic, some not, with one overload per channel flavour and two additional overloads for arrays of asynchronous channels. We resort to an `out` parameter simply to simulate overloading on return type, which is, unfortunately, illegal in C#. Although distasteful, overloading in this way means that boilerplate calls to `Initialize(out channel)` do not have to be altered when changing the flavour of `channel`.

The `When` method begins the construction of a new join pattern and like `Initialize`, has eight overloads, one per channel flavour and two more for arrays of asynchronous channels. The return type of `When` is invariably some instance of the class scheme:

$$\texttt{JoinPattern}[\langle R\rangle].\texttt{OpenPattern}[\langle A|A[]\rangle]$$

Here R is the optional return type of a synchronous pattern and A is the optional argument type of the channel or channel array.

There are two flavours of `JoinPattern`. The non-generic `JoinPattern` class contains nested `OpenPattern` classes whose continuations all return `void`. The generic `JoinPattern<R>` family of classes contains nested `OpenPattern` classes whose continuations all return `R`. More precisely, each `JoinPattern` family contains $max + 1$ nested subclasses, $\texttt{OpenPattern}\langle P_1, \ldots, P_n\rangle$ $(0 \leq \ n \leq max)$, each overloaded on generic arity n:

```
abstract class JoinPattern[<R>] {
  class OpenPattern: JoinPattern[<R>] { ...}
  class OpenPattern<P1>: JoinPattern[<R>] { ...}
  ⋮
  class OpenPattern<P1, ..., Pmax>: JoinPattern[<R>] { ...} }
```

In turn, each $\texttt{OpenPattern}\langle P_1, \ldots, P_n\rangle$ class has the schematic form:

```
class OpenPattern<P1,...,Pn> : JoinPattern[<R>] {
  delegate [void | R] Continuation(P1 p1,...,Pn pn);
  void Do(Continuation continuation);
  OpenPattern<P1,...,Pn> And(Asynchronous.Channel c);
  OpenPattern<P1,...,Pn> And(Asynchronous.Channel[] cs);
  OpenPattern<P1,...,Pn,Pn+1> And<Pn+1>(
    Asynchronous.Channel<Pn+1> c);        (n < max)
  OpenPattern<P1,...,Pn,Pn+1[]> And<Pn+1>(
    Asynchronous.Channel<Pn+1>[] cs);     (n < max)
  OpenPattern<P1,...,Pair<Pn, Pn+1>> AndPair<Pn+1>(
    Asynchronous.Channel<Pn+1> c);        (n > 0)
  OpenPattern<P1,...,Pair<Pn,Pn+1[]>> AndPair<Pn+1>(
    Asynchronous.Channel<Pn+1>[] cs);     (n > 0)
}
```

Class `OpenPattern`$\langle P_1, \ldots, P_n \rangle$ declares its own nested `Continuation` delegate type taking invocation arguments $p_1, \ldots, p_n$ of types $P_1, \ldots, P_n$ and returning `void` or R, as appropriate. The `And` and `AndPair` methods with side conditions on n are only included for satisfying n. The class declares up to four overloads of method `And`, two generic, two non-generic, one for each flavour of asynchronous channel and one for each array thereof. A non-generic `And` method constructs a new open pattern of the same type (and thus expecting the same type of `Continuation`) as `this`, that synchronizes with an additional (data-less) channel or set thereof. A generic `And`$<P_{n+1}>$ method on `OpenPattern`$\langle P_1, \ldots, P_n \rangle$ constructs a new successor pattern of type `OpenPattern`$\langle P_1, \ldots, P_n, P \rangle$, thus binding one additional continuation type and argument. Type P is P_{n+1} or $P_{n+1}[]$, if the argument is a single channel, `c`, or array of channels, `cs`. The `AndPair`$<P_{n+1}>$ methods use pairing to avoid introducing another continuation argument: in particular, for $n = \max$, calling `AndPair` is the only way to extend the pattern to wait for additional data-carrying channels.

Every `JoinPattern` contains an instance of an internal class `Pattern`, which represents a conjunction of atomic patterns (channels or channel arrays), as a tree. `Pattern`'s `GetIntSet` method computes the summary `IntSet` used for scheduling which is all the `Join` scheduler needs to know to select a pattern for execution; it also does some error checking. `Pattern` has these subclasses (omitting similar ones for synchronous channels and channel arrays):

```
abstract class Pattern { S GetIntSet<S>(...) where S: IIntSet<S>; }
abstract class Pattern<P> : Pattern { abstract P Get(); }
class Atom: Pattern<Unit> { Atom(Asynchronous.Channel c); ... }
class Atom<A>: Pattern<A> { Atom(Asynchronous.Channel<A> c); ... }
class And<Q,R>: Pattern<Pair<Q,R>>
 { And(Pattern<Q> fst, Pattern<R> snd); ... }
class And<Q>: Pattern<Q> { And(Pattern<Q> fst, Pattern<Unit> snd); ... }
```

Subclass `Pattern<P>` of `Pattern` hides an existential type `P`, the return type of its abstract method `P Get()`. Method `Get` is used to dequeue all of a pattern's channels, returning a single, composite value of their queue heads. `Get`

is interesting because, due to base class specialization, its return type actually varies with each concrete subclass: `new Atom(c).Get()` returns `P=Unit` (`Unit` is an empty struct with one value); `new Atom<A>(c).Get()` returns `P=A`; `new And<Q,R>(fst,snd).Get()` returns `P=Pair<Q,R>` (a struct with two fields) and `new And<Q>(fst,snd).Get()` returns `P=Q`, absorbing the data-less `Unit`-pattern `snd`. Technically, the hierarchy rooted at `Pattern` is a simple instance of a *Generalized Algebraic Datatype (GADT)* [6]. When a `JoinPattern` is selected for execution, a virtual method `Fire()` or `Spawn()`, declared on `JoinPattern`, but overridden in each `OpenPattern`$\langle P_1, \ldots, P_n \rangle$ subclass, calls `pattern.Get()` on a private field, `pattern`, of specialized type `And`$\langle$`Pair`$\langle \ldots$ `Pair`$\langle P_1, \ldots \rangle, \ldots \rangle, P_n \rangle$. This yields a nested pair of n-components of the appropriate type to pass on, component-wise, to its n-ary `Continuation`. This is quite elegant since no boxing, heap allocation or casting is required to implement the dequeuing and transfer of multiple values. The `And` method of an `OpenPattern` extends its current `pattern` by conjoining it with a new atomic pattern; `AndPair` extends its current `pattern` - a conjunction - by conjoining its first component with the conjunction of its second component and a new atomic pattern.

Calling `When` allocates a new `OpenPattern` with an atomic `pattern` field and null continuation. The `OpenPattern` contains another field storing a callback to invoke with a `JoinPattern` when the `OpenPattern` is supplied with a continuation. Calling `And/AndPair` returns a new `OpenPattern` with an extended `pattern`, same callback and null continuation. Calling `Do` creates a new `OpenPattern` with the same `pattern`, null callback and non-null continuation and passes it, as a `JoinPattern`, to the original callback. The callback finally grabs the `Join` lock, calls `GetIntSet` and either detects an illegal pattern or inserts an entry into the appropriate lists (`Actions` and perhaps `Patterns`).

6 Conclusion and Related Work

Compared with `Joins`, Cω offers more static checking, e.g. rejecting non-linear patterns, and much better error messages. It also has more opportunities for optimization: Cω could use static analysis to determine whether an asynchronous continuation can safely be run in the enabling thread, rather than a new one. Cω knows the methods and patterns belonging to a class and can thus compile pattern matching as a cascading test of the state against pre-computed bitmask constants, with the scheduling code shared between all instances of the class; `Joins` must instead perform a linear search through a heap-allocated, unshared list of patterns, (re-)constructed for each `Join` instance. Cω can also inline all the continuations of a synchronous method into its compiled body, instead of indirecting through delegates. On one micro-benchmark, pitting a Cω `OnePlaceBuffer` against a `Joins` implementation, we found that allocating 1000 buffers in a tight loop is roughly 60x slower with `Joins`, due to the overhead of reconstructing the patterns for each buffer; executing 1000 `Put` then `Get` calls in the same thread is 2x slower, reflecting the cost of indirecting through a channel delegate and consulting the heap-allocated patterns; but the time needed

to run a producer and consumer thread exchanging 1000 messages is roughly comparable, with any differences dominated by the cost of context switching.

The join calculus [7] provides the foundation for join patterns. JoCaml [8] and Funnel [9] are functional languages supporting declarative join patterns. The CCR [5] is an asynchronous concurrency library for C# that uses custom scheduling rather than integrating with the host's thread API as `Joins` does. The CCR supports join patterns, but not synchronous ones; programs must be written in an awkward continuation passing style, alleviated sometimes by the use of C# iterators. Singh [10] builds an experimental combinator library for joins patterns using software transactional memory in STM Haskell but the implementation is more expository than practical due to performance issues.

Future avenues to explore include supporting Ada-style synchronous rendezvous, allowing more than one synchronous channel to occur in a pattern. Executing asynchronous patterns in a new thread is expensive and not always required: if the continuation is non-blocking and guaranteed to return quickly, it can be executed immediately in the thread that enabled the pattern. Adapting `Joins` to support such user-controlled scheduling of asynchronous patterns is straightforward and has other applications, for instance to queue continuations in a thread pool or in the event loop of a GUI thread. A library makes such experimentation much easier.

References

1. Microsoft Research: Cω, http://research.microsoft.com/Comega (2004)
2. Russo, C.: Joins: A Concurrency Library (2006) Binaries with tutorial and samples: http://research.microsoft.com/research/downloads.
3. Benton, N., Cardelli, L., Fournet, C.: Modern concurrency abstractions for C#. ACM Transactions on Programming Languages and Systems **26** (2004)
4. Itzstein, G.S., Kearney, D.: Join Java: An alternative concurrency semantics for Java. Technical Report ACRC-01-001, University of South Australia (2001)
5. Chrysanthakopoulos, G., Singh, S.: An asynchronous messaging library for C#. In: Synchronization and Concurrency in Object-Oriented Languages (SCOOL), OOPSLA 2005 Workshop, UR Research (2005)
6. Kennedy, A.J., Russo, C.V.: Generalized algebraic data types and object-oriented programming. In: Object-Oriented Programming: Systems, Languages, Applications (OOPSLA), San Diego, ACM (2005)
7. Fournet, C., Gonthier, G.: The join calculus: a language for distributed mobile programming. In: APPSEM Summer School, Caminha, Portugal, September 2000. Volume 2395 of LNCS., Springer-Verlag (2002)
8. Fournet, C., Le Fessant, F., Maranget, L., Schmitt, A.: JoCaml: a language for concurrent distributed and mobile programming. In: Advanced Functional Programming, 4th International School, Oxford, August 2002. Volume 2638 of LNCS., Springer-Verlag (2003)
9. Odersky, M.: An overview of functional nets. In: APPSEM Summer School, Caminha, Portugal, September 2000. Volume 2395 of LNCS., Springer-Verlag (2002)
10. Singh, S.: Higher-order combinators for join patterns using STM. TRANSACT ACM Workshop on Languages, Compilers and Hardware Support for Transactional Computing (2006)