Channels – the CSP/CML Model

Radu Nicolescu
Department of Computer Science
University of Auckland

28 March 2019
31 March 2019
1. CSP/CML
2. Hopac
3. C# Channels
4. Counter Sample
5. Compile and Run
6. State Sample
7. Async Post
Outline

1. CSP/CML
2. Hopac
3. C# Channels
4. Counter Sample
5. Compile and Run
6. State Sample
7. Async Post
• Communicating sequential processes (CSP) is a formal language for describing patterns of interaction in concurrent systems
  • inspired channels, rendezvous in programming
  • family of process calculi/algebras: CSP, CCS, $\pi$, join, ...

• CSP was first described in a 1978 paper by Tony Hoare, but has since evolved substantially

• Tony Hoare
  • algorithm: quicksort
  • formal language: CSP
  • concurrency: monitors, dining philosophers
  • programming languages: occam language, Algol W
  • null = billion dollar mistake!
CSP (wiki)

- Communicating sequential processes (CSP) is a formal language for describing patterns of interaction in concurrent systems
  - inspired channels, rendezvous in programming
  - family of process calculi/algebras: CSP, CCS, π, join, ...
- CSP was first described in a 1978 paper by Tony Hoare, but has since evolved substantially
  - Tony Hoare
    - algorithm: quicksort
    - formal language: CSP
    - concurrency: monitors, dining philosophers
    - programming languages: occam language, Algol W
    - null = billion dollar mistake!
CSP (wiki)

- **Communicating sequential processes (CSP)** is a formal language for describing patterns of interaction in concurrent systems
  - inspired *channels, rendezvous* in programming
  - family of *process calculi/algebras*: CSP, CCS, π, join, ...

- **CSP** was first described in a 1978 paper by **Tony Hoare**, but has since evolved substantially

- **Tony Hoare**
  - algorithm: quicksort
  - formal language: CSP
  - concurrency: monitors, dining philosophers
  - programming languages: occam language, Algol W
  - **null** = billion dollar mistake!
CML (wiki)

- First functional language: **LISP** (John McCarthy, 1958) = List Processor (Lots of Insipid Stupid Parentheses 😊)

- **LISP** family: LISP, Scheme, Clojure... JavaScript!

- **ML** family: ML (Meta Language), Standard ML, CAML (Categorical Abstract Machine Language), OCAML (Object CAML), Scala, **F#** (OCAML for .NET)

- John Reppy CML (Concurrent ML): channels, rendezvous, events from **CSP**

- Channels: CML (language), Go (language), F# Hopac (library), C# Channels (library), JavaScript, Rust, Kotlin, ...

- Rendezvous, Joins: Ada, JoCAML (Joins CAML), Cω, ...
CML (wiki)

- **First functional language**: LISP (John McCarthy, 1958) = List Processor (Lots of Insipid Stupid Parentheses 😊)

- **LISP family**: LISP, Scheme, Clojure… JavaScript!

- **ML family**: ML (Meta Language), Standard ML, CAML (Categorical Abstract Machine Language), OCAML (Object CAML), Scala, F# (OCAML for .NET)

- John Reppy CML (Concurrent ML): channels, rendezvous, events from CSP

- **Channels**: CML (language), Go (language), F# Hopac (library), C# Channels (library), JavaScript, Rust, Kotlin, …

- Rendezvous, Joins: Ada, JoCAML (Joins CAML), Cω, …
CML (wiki)

- First functional language: LISP (John McCarthy, 1958) = List Processor (Lots of Insipid Stupid Parentheses 😊)
- LISP family: LISP, Scheme, Clojure... JavaScript!
- ML family: ML (Meta Language), Standard ML, CAML (Categorical Abstract Machine Language), OCAML (Object CAML), Scala, F# (OCAML for .NET)
- John Reppy CML (Concurrent ML): channels, rendezvous, events from CSP
- Channels: CML (language), Go (language), F# Hopac (library), C# Channels (library), JavaScript, Rust, Kotlin, ...
- Rendezvous, Joins: Ada, JoCAML (Joins CAML), Cω, ...
CML (wiki)

- First functional language: **LISP** (John McCarthy, 1958) = List Processor (Lots of Insipid Stupid Parentheses 😊)

- **LISP** family: LISP, Scheme, Clojure... JavaScript!

- **ML** family: ML (Meta Language), Standard ML, CAML (Categorical Abstract Machine Language), OCAML (Object CAML), Scala, F# (OCAML for .NET)

- **John Reppy** CML (Concurrent ML): channels, rendezvous, events from CSP

- Channels: CML (language), Go (language), F# Hopac (library), C# Channels (library), JavaScript, Rust, Kotlin, ...

- Rendezvous, Joins: Ada, JoCAML (Joins CAML), Cω, ...
CML (wiki)

- First functional language: LISP (John McCarthy, 1958) = List Processor (Lots of Insipid Stupid Parentheses 😊)
- LISP family: LISP, Scheme, Clojure… JavaScript!
- ML family: ML (Meta Language), Standard ML, CAML (Categorical Abstract Machine Language), OCAML (Object CAML), Scala, F# (OCAML for .NET)
- John Reppy CML (Concurrent ML): channels, rendezvous, events from CSP
- Channels: CML (language), Go (language), F# Hopac (library), C# Channels (library), JavaScript, Rust, Kotlin, ...
- Rendezvous, Joins: Ada, JoCAML (Joins CAML), Cω, …
Channels – bird’s eye view

- **Named communication channels** between concurrent tasks

  ![Diagram of channels](image)

- Default: multiple writers, aka multiple readers
- Default: zero-size buffer, aka rendezvous (sync handshake)
- Extensions: bounded buffers, even unbounded (≈ actors)
- Many lightweight async tasks (aka green threads)
- Theory: channels ≡ actors (but shine for different apps)
Outline

1. CSP/CML
2. Hopac
3. C# Channels
4. Counter Sample
5. Compile and Run
6. State Sample
7. Async Post
Hopac Readings

- Hopac repository
  https://github.com/Hopac/Hopac


- Hopac Reference – monads, high-order combinatorics...
  https://hopac.github.io/Hopac/Hopac.html

- Demistify FP – Hopac
  https://www.demystifyfp.com/tags/hopac/
Performance – Hopac vs Akka

Stream performance – the lower the better
http://vaskir.blogspot.com/2016/05/akkanet-streams-vs-hopac.html
Actor-like performance – the higher the better

https://vasily-kirichenko.github.io/fsharpblog/actors
Hopac Basics

- Create a default typed channel

```csharp
let ch = Ch<string>()
```

- A job = lightweight async task (as a sugared monad) here w/ async sleep

```csharp
job {
    do! timeOutMillis 100
}
```

- Communicate via a channel

```csharp
job {
    do! ch *= m // do! Ch.give ch m // sync post
    do! ch *=+ m // do! Ch.send ch m // async post
    let! m = ch // let! m = Ch.take ch // receive
}
```
Outline

1. CSP/CML
2. Hopac
3. C# Channels
4. Counter Sample
5. Compile and Run
6. State Sample
7. Async Post
C# Channels Readings

- MSDN System.Threading.Channels Namespace
  https://docs.microsoft.com/en-us/dotnet/api/system.threading.channels?view=dotnet-plat-ext-2.1

- Github System.Threading.Channels

- Exploring System.Threading.Channels
  https://ndportmann.com/system-threading-channels/
C# Channels Basics

- Create a size 1 one-to-one typed channel (no size 0)

```csharp
var opt = new BoundedChannelOptions(1)
{
    SingleWriter=true, SingleReader=true,
};
var ch = Channel.CreateBounded<string>(opt);
```

- Use the existing async/await framework
- Communicate via a channel

```csharp
async Task ...
{
    await ch.Writer.WriteAsync(m); // async post

    var m = await ch.Reader.ReadAsync(); // receive
}
```
Outline

1. CSP/CML
2. Hopac
3. C# Channels
4. Counter Sample
5. Compile and Run
6. State Sample
7. Async Post
Counter – F# Mailbox

- actor = inbox + async function

```fsharp
let agent = MailboxProcessor.Start(fun inbox ->
    let rec loop count =
        async {
            let! msg = inbox.Receive()
            do! Async.Sleep 100
            return! loop (count + 1)
        }
    loop 0
)
```

- easy to post message from outside

```fsharp
agent.Post m
```
Counter – F# Hopac

- separate channel, job (async-like) function

```fsharp
let ch = Ch<string>()

let agent =
    let rec loop count = job {
        let! msg = ch // let! msg = Ch.take ch
        do! timeOutMillis 100
        return! loop (count + 1)
    }
start (loop 0)
```

- easier to post message from another job, that could run sync

```fsharp
let setup = job {
    do! ch <- m // do! Ch.give ch m
}

run setup
```
Counter – C# Channels

• separate channel (w/ options) =

```csharp
BoundedChannelOptions one2one =
    new BoundedChannelOptions (1)
    {
        SingleWriter = true,
        SingleReader = true,
    };

Channel<string> ch =
    Channel.CreateBounded<string> (one2one);
```

• separate async function (not started)

```csharp
async Task agent () {
    var count = 0;
    for (; ; ) {
        var msg = await ch.Reader.ReadAsync ();
        count += 1;
        await Task.Delay (100);
    }
}
```
Counter – C# Channels

- easier to post message from another async task

```csharp
async Task Main() {
    var a = agent(); // create agent and start it async
    await ch.Writer.WriteAsync(m);
}
```
Outline

1. CSP/CML
2. Hopac
3. C# Channels
4. Counter Sample
5. Compile and Run
6. State Sample
7. Async Post
Compile and Run – F# Mailbox

- Mailbox in FSharp.Core, so no special configuration required
- All F# code: Linqpad preamble ⇔ module line in .FS

```
<Query Kind="FSharpProgram" />

module M
```

- Command-line compilation

```
fsc F#-Actor-counters.fs
```

- For all F# programs: you may also want to create a proper Main function, that will invoke the rest

```
[<EntryPoint>]
let main args =
  ...
0
```
Compile and Run – F# Hopac

- Hopac NOT in FSharp.Core, additional configuration required

- Command-line compilation – two libraries

  1. `fsc -r:Hopac.dll -r:Hopac.Core.dll`
  2. `F#–Hopac–counters.fs`

- Runtime – one more library

  1. `Hopac.Platform.dll`

- Runtime – F#-Hopac-counters.exe.config, to ensure server garbage collection

```xml
<configuration>
  <runtime>
    <gcServer enabled="true"/>
  </runtime>
</configuration>
```
Compile and Run – C# Channels

- Channels lib NOT in system, additional configuration required
- Command-line compilation – three libraries + netstandard

```
csc  -r:mscorlib.dll, netstandard.dll,
    System.Threading.Channels.dll,
    System.Threading.Tasks.Extensions.dll,
C#-Channels-counters.cs
```

- Runtime – one more library

```
System.Collections.Immutable.dll
```

- Code – see and compare the samples

```
static void Main (string[] args) {
    new Program().Main2().Wait();
}
```
Outline

1. CSP/CML
2. Hopac
3. C# Channels
4. Counter Sample
5. Compile and Run
6. State Sample
7. Async Post
Outline

1. CSP/CML
2. Hopac
3. C# Channels
4. Counter Sample
5. Compile and Run
6. State Sample
7. Async Post
Hopac Async Post

- **Async** communication via a **0-size** channel?

```csharp
job {
    do! ch *<- m // do! Ch.give ch m // sync post

    do! ch *<+ m // do! Ch.send ch m // async post
}
```

- Ch.give is a sync post: the writer awaits (logically suspended) until the rendezvous!
- Ch.send is an async post: the writer generates a hidden helper job that takes charge of the actual writing!
- this helper job awaits until the rendezvous! Cf. code sample!
- thus the original writer job can continue!
Hopac Async Post

- **Async** communication via a 0-size channel?

```plaintext
job {
  do! ch *<− m // do! Ch.give ch m  // sync post
  do! ch *<+ m // do! Ch.send ch m  // async post
}
```

- **Ch.give** is a **sync post**: the writer **awaits** (logically suspended) until the rendezvous!

- **Ch.send** is an **async post**: the writer generates a hidden helper job that takes charge of the actual writing!

- this helper job **awaits** until the rendezvous! Cf. code sample!

- thus the original writer job can **continue**!
Hopac Async Post

- **Async** communication via a 0-size channel?

```csharp
job {    
    do! ch *<- m // do! Ch.give ch m // sync post
}
```

- **Ch.give** is a **sync post**: the writer **awaits** (logically suspended) until the rendezvous!

- **Ch.send** is an **async post**: the writer generates a hidden helper job that takes charge of the actual writing!

- this helper job **awaits** until the rendezvous! Cf. code sample!

- thus the original writer job can **continue**!
Simulating async send on sync give

```plaintext
let ( *<++ ) ch m = // ch *<+ m
    job {
        let help = job {
            do! ch *<-- m
        }
        start help
    }
```

Usage – awaiting for help start (not completion)

```plaintext
do! ch *<++ m  // do! ch *<+ m
```
## Async Post Samples – out of the box features

<table>
<thead>
<tr>
<th></th>
<th>size</th>
<th>sync write</th>
<th>async write</th>
<th>read</th>
</tr>
</thead>
<tbody>
<tr>
<td>F# Hopac Chan</td>
<td>= 0</td>
<td>*&lt;- .give</td>
<td>*&lt;&lt;+ .send</td>
<td>.take</td>
</tr>
<tr>
<td>“ BoundedMb</td>
<td>≥ 0</td>
<td>.put</td>
<td></td>
<td>.take</td>
</tr>
<tr>
<td>“ Mailbox</td>
<td>∞</td>
<td></td>
<td>.send</td>
<td>.take</td>
</tr>
<tr>
<td>F# Actors</td>
<td>∞</td>
<td></td>
<td>.Post</td>
<td>.Receive</td>
</tr>
<tr>
<td>C# Channels</td>
<td>≥ 1, ∞</td>
<td></td>
<td>.WriteAsync</td>
<td>.ReadAsync</td>
</tr>
<tr>
<td>GO Channels</td>
<td>≥ 0</td>
<td></td>
<td></td>
<td>&lt;−</td>
</tr>
<tr>
<td>JS Channels</td>
<td>≥ 0</td>
<td></td>
<td>.push</td>
<td>.shift</td>
</tr>
</tbody>
</table>

More later: backpressure, select, ...