Concurrent ML

John Reppy
jhr@cs.uchicago.edu

University of Chicago

January 21, 2016

Outline

- Concurrent programming models
- Concurrent ML
- Multithreading via continuations (if there is time)

Outline

- Concurrent programming models
- Concurrent ML
- Multithreading via continuations (if there is time)

Outline

- Concurrent programming models
- Concurrent ML
- Multithreading via continuations (if there is time)

Outline

- Concurrent programming models
- Concurrent ML
- Multithreading via continuations (if there is time)

Different language-design axes

- Parallel vs. concurrent vs. distributed.
- Implicitly parallel vs. implicitly threaded vs. explicitly threaded.
- Deterministic vs. non-deterministic.
- Shared state vs. shared-nothing.

In this lecture, I will mostly focus on explicitly-threaded, non-deterministic, shared-nothing, concurrent languages

Different language-design axes

- Parallel vs. concurrent vs. distributed.
- Implicitly parallel vs. implicitly threaded vs. explicitly threaded.
- Deterministic vs. non-deterministic.
- Shared state vs. shared-nothing.

In this lecture, I will mostly focus on explicitly-threaded, non-deterministic, shared-nothing, concurrent languages

Different language-design axes

- Parallel vs. concurrent vs. distributed.
- Implicitly parallel vs. implicitly threaded vs. explicitly threaded.
- Deterministic vs. non-deterministic.
- Shared state vs. shared-nothing.

In this lecture, I will mostly focus on explicitly-threaded, non-deterministic, shared-nothing, concurrent languages

Different language-design axes

- Parallel vs. concurrent vs. distributed.
- Implicitly parallel vs. implicitly threaded vs. explicitly threaded.
- Deterministic vs. non-deterministic.
- Shared state vs. shared-nothing.

In this lecture, I will mostly focus on explicitly-threaded, non-deterministic, shared-nothing, concurrent languages

Different language-design axes

- Parallel vs. concurrent vs. distributed.
- Implicitly parallel vs. implicitly threaded vs. explicitly threaded.
- Deterministic vs. non-deterministic.
- Shared state vs. shared-nothing.

In this lecture, I will mostly focus on explicitly-threaded, non-deterministic, shared-nothing, concurrent languages

Parallelism vs. concurrency

Parallel and concurrent programming address two different problems.

- Parallelism is about speed — exploiting parallel processors to solve problems quicker.
- Concurrency is about nondeterminism — managing the unpredictable external world.

Parallelism vs. concurrency

Parallel and concurrent programming address two different problems.

- Parallelism is about speed — exploiting parallel processors to solve problems quicker.
- Concurrency is about nondeterminism — managing the unpredictable external world.

Parallelism vs. concurrency

Parallel and concurrent programming address two different problems.

- Parallelism is about speed — exploiting parallel processors to solve problems quicker.
- Concurrency is about nondeterminism — managing the unpredictable external world.

Why concurrency?

- Many applications are reactive systems that must cope with non-determinism (*e.g.*, users and the network).
- Concurrency provides a clean abstraction of such interactions by hiding the underlying interleaving of execution.
- Thread abstraction is useful for large-grain, heterogeneous parallelism.

Why concurrency?

- Many applications are reactive systems that must cope with non-determinism (*e.g.*, users and the network).
- Concurrency provides a clean abstraction of such interactions by hiding the underlying interleaving of execution.
- Thread abstraction is useful for large-grain, heterogeneous parallelism.

Why concurrency?

- Many applications are reactive systems that must cope with non-determinism (*e.g.*, users and the network).
- Concurrency provides a clean abstraction of such interactions by hiding the underlying interleaving of execution.
- Thread abstraction is useful for large-grain, heterogeneous parallelism.

Why concurrency?

- Many applications are reactive systems that must cope with non-determinism (*e.g.*, users and the network).
- Concurrency provides a clean abstraction of such interactions by hiding the underlying interleaving of execution.
- Thread abstraction is useful for large-grain, heterogeneous parallelism.

Synchronization and communication

For concurrent languages, the choice of synchronization and communication mechanisms is critical.

- Should these be independent or coupled?
- What guarantees should be provided?

Synchronization and communication

For concurrent languages, the choice of synchronization and communication mechanisms is critical.

- Should these be independent or coupled?
- What guarantees should be provided?

Synchronization and communication

For concurrent languages, the choice of synchronization and communication mechanisms is critical.

- Should these be independent or coupled?
- What guarantees should be provided?

Concurrency is hard(?)

Concurrent programming has a reputation of being hard.

- The problem is that shared-memory concurrency using locks and condition variables is the dominant model in concurrent languages.
- Shared-memory programming requires a defensive approach: protect against data races.
- Synchronization and communication are decoupled.
- Shared state often leads to poor modularity.

Concurrency is hard(?)

Concurrent programming has a reputation of being hard.

- The problem is that shared-memory concurrency using locks and condition variables is the dominant model in concurrent languages.
- Shared-memory programming requires a defensive approach: protect against data races.
- Synchronization and communication are decoupled.
- Shared state often leads to poor modularity.

Concurrency is hard(?)

Concurrent programming has a reputation of being hard.

- The problem is that shared-memory concurrency using locks and condition variables is the dominant model in concurrent languages.
- Shared-memory programming requires a defensive approach: protect against data races.
- Synchronization and communication are decoupled.
- Shared state often leads to poor modularity.

Concurrency is hard(?)

Concurrent programming has a reputation of being hard.

- The problem is that shared-memory concurrency using locks and condition variables is the dominant model in concurrent languages.
- Shared-memory programming requires a defensive approach: protect against data races.
- Synchronization and communication are decoupled.
- Shared state often leads to poor modularity.

Concurrency is hard(?)

Concurrent programming has a reputation of being hard.

- The problem is that shared-memory concurrency using locks and condition variables is the dominant model in concurrent languages.
- Shared-memory programming requires a defensive approach: protect against data races.
- Synchronization and communication are decoupled.
- Shared state often leads to poor modularity.

Software transactional memory

- Software transactional memory (STM) has been offered as a solution.
- Ideal semantics is appealing: simple and intuitive.
- Reality is less so. Issues of nesting, exceptions, I/O, weak vs. strong atomicity, make things much more complicated.
- Also, STM does support conditional synchronization well.

Software transactional memory

- Software transactional memory (STM) has been offered as a solution.
- Ideal semantics is appealing: simple and intuitive.
- Reality is less so. Issues of nesting, exceptions, I/O, weak vs. strong atomicity, make things much more complicated.
- Also, STM does support conditional synchronization well.

Software transactional memory

- Software transactional memory (STM) has been offered as a solution.
- Ideal semantics is appealing: simple and intuitive.
- Reality is less so. Issues of nesting, exceptions, I/O, weak vs. strong atomicity, make things much more complicated.
- Also, STM does support conditional synchronization well.

Software transactional memory

- Software transactional memory (STM) has been offered as a solution.
- Ideal semantics is appealing: simple and intuitive.
- Reality is less so. Issues of nesting, exceptions, I/O, weak vs. strong atomicity, make things much more complicated.
- Also, STM does support conditional synchronization well.

Software transactional memory

- Software transactional memory (STM) has been offered as a solution.
- Ideal semantics is appealing: simple and intuitive.
- Reality is less so. Issues of nesting, exceptions, I/O, weak vs. strong atomicity, make things much more complicated.
- Also, STM does support conditional synchronization well.

Message passing

In 1978, Tony Hoare proposed a concurrent programming model based on independent processes that communicate via messages (CSP).

- Well-defined interfaces between independent, sequential, components.
- Natural encapsulation of state.
- Extends more easily to distributed implementation.
- Inspired many language designs, including CML, go (and its predecessors), OCCAM, OCCAM-π, *etc.*.

Message passing

In 1978, Tony Hoare proposed a concurrent programming model based on independent processes that communicate via messages (CSP).

- Well-defined interfaces between independent, sequential, components.
- Natural encapsulation of state.
- Extends more easily to distributed implementation.
- Inspired many language designs, including CML, go (and its predecessors), OCCAM, OCCAM-π, *etc.*.

Message passing

In 1978, Tony Hoare proposed a concurrent programming model based on independent processes that communicate via messages (CSP).

- Well-defined interfaces between independent, sequential, components.
- Natural encapsulation of state.
- Extends more easily to distributed implementation.
- Inspired many language designs, including CML, go (and its predecessors), OCCAM, OCCAM-π, *etc.*.

Message passing

In 1978, Tony Hoare proposed a concurrent programming model based on independent processes that communicate via messages (CSP).

- Well-defined interfaces between independent, sequential, components.
- Natural encapsulation of state.
- Extends more easily to distributed implementation.
- Inspired many language designs, including CML, go (and its predecessors), OCCAM, OCCAM-π, *etc.*.

Message passing

In 1978, Tony Hoare proposed a concurrent programming model based on independent processes that communicate via messages (CSP).

- Well-defined interfaces between independent, sequential, components.
- Natural encapsulation of state.
- Extends more easily to distributed implementation.
- Inspired many language designs, including CML, go (and its predecessors), OCCAM, OCCAM-π, *etc.*.

Message-passing design space

- Synchronous vs. asynchronous vs. RPC-style communication.
- Per-thread message addressing vs. channels
- Synchronization constructs: asymmetric choice, symmetric choice, join-patterns.

Message-passing design space

- Synchronous vs. asynchronous vs. RPC-style communication.
- Per-thread message addressing vs. channels
- Synchronization constructs: asymmetric choice, symmetric choice, join-patterns.

Message-passing design space

- Synchronous vs. asynchronous vs. RPC-style communication.
- Per-thread message addressing vs. channels
- Synchronization constructs: asymmetric choice, symmetric choice, join-patterns.

Message-passing design space

- Synchronous vs. asynchronous vs. RPC-style communication.
- Per-thread message addressing vs. channels
- Synchronization constructs: asymmetric choice, symmetric choice, join-patterns.

Channels

For the rest of the talk, we assume channel-based communication with synchronous message passing.
In SML, we can define the following interface to this model:

```
type 'a chan

val channel : unit -> 'a chan
val recv : 'a chan -> 'a
val send : ('a chan * 'a) -> unit
```

We might also include a way to monitor multiple channels, such as the following asymmetric choice operator:

```
val selectRecv : ('a chan * ('a -> 'b)) list -> 'b
```

Interprocess communication

In practice, it is often the case that

- interactions between processes involve multiple messages.
- processes need to interact with multiple partners (nondeterministic choice).

These two properties of IPC cause a conflict.

Interprocess communication

In practice, it is often the case that

- interactions between processes involve multiple messages.
- processes need to interact with multiple partners (nondeterministic choice).

These two properties of IPC cause a conflict.

Interprocess communication

In practice, it is often the case that

- interactions between processes involve multiple messages.
- processes need to interact with multiple partners (nondeterministic choice).

These two properties of IPC cause a conflict.

Interprocess communication

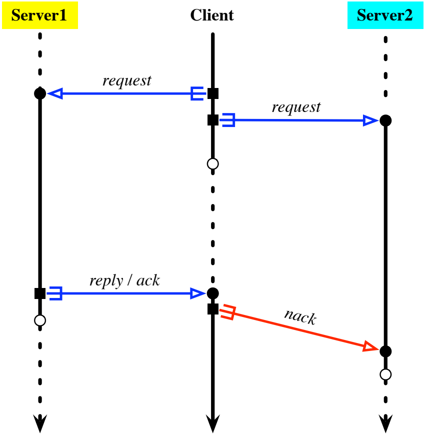
In practice, it is often the case that

- interactions between processes involve multiple messages.
- processes need to interact with multiple partners (nondeterministic choice).

These two properties of IPC cause a conflict.

Interprocess communication *(continued ...)*

For example, consider a possible interaction between a client and two servers.



Interprocess communication *(continued ...)*

Without abstraction, the code is a mess.

```
let  val replCh1 = channel() and nack1 = cvar()
     val replCh2 = channel() and nack2 = cvar()
in
   send (reqCh1, (req1, replCh1, nack1));
   send (reqCh2, (req2, replCh2, nack2));
   selectRecv [
     (replCh1, fn repl1 => ( set nack2;  act1 repl1),
     (replCh2, fn repl2 => ( set nack1;  act2 repl2)
   ]
end
```

But traditional abstraction mechanisms do not support choice!

Concurrent ML

- Provides a uniform framework for synchronization: *events*.
- Event combinators for constructing abstract protocols.
- Collection of event constructors:
  - I-variables
  - M-variables
  - Mailboxes
  - Channels

Plus I/O, timeouts, thread join, ...

Concurrent ML

- Provides a uniform framework for synchronization: *events*.
- Event combinators for constructing abstract protocols.
- Collection of event constructors:
  - I-variables
  - M-variables
  - Mailboxes
  - Channels

Plus I/O, timeouts, thread join, ...

Concurrent ML

- Provides a uniform framework for synchronization: *events*.
- Event combinators for constructing abstract protocols.
- Collection of event constructors:
  - I-variables
  - M-variables
  - Mailboxes
  - Channels

Plus I/O, timeouts, thread join, ...

Concurrent ML

- Provides a uniform framework for synchronization: *events*.
- Event combinators for constructing abstract protocols.
- Collection of event constructors:
  - I-variables
  - M-variables
  - Mailboxes
  - Channels

  Plus I/O, timeouts, thread join, ...

Events

- We use event values to package up protocols as first-class abstractions.
- An event is an abstraction of a synchronous operation, such as receiving a message or a timeout.

```
type 'a event
```

- Base-event constructors create event values for communication primitives:

```
val recvEvt : 'a chan -> 'a event
val sendEvt : 'a chan -> unit event
```

Events *(continued ...)*

Event operations:

- Event wrappers for post-synchronization actions:

```
val wrap : ('a event * ('a -> 'b)) -> 'b event
```

- Event generators for pre-synchronization actions and cancellation:

```
val guard    : (unit -> 'a event) -> 'a event
val withNack : (unit event -> 'a event) -> 'a event
```

- Choice for managing multiple communications:

```
val choose : 'a event list -> 'a event
```

- Synchronization on an event value:

```
val sync : 'a event -> 'a
```

Swap channels

A swap channel is an abstraction that allows two threads to swap values.

```
type 'a swap_chan

val swapChannel : unit -> 'a swap_chan
val swapEvt     : 'a swap_chan * 'a -> 'a event
```

Swap channels *(continued ...)*

The basic implementation of swap channels is straightforward.

```
datatype 'a swap_chan = SC of ('a * 'a chan) chan

fun swapChannel () = SC(channel ())

fun swap (SC ch, vOut) = let
      val inCh = channel ()
      in
        select [
            wrap (recvEvt ch, fn (vIn, outCh) => (send(outCh, vOut); vIn)),
            wrap (sendEvt (ch, (vOut, inCh)), fn () => recv inCh)
          ]
      end
```

Note that the `swap` function both offers to send and receive on the channel so as to avoid deadlock.

Making swap channels first class

We can also make the `swap` operation first class

```
val swapEvt : 'a swap_chan * 'a -> 'a event
```

by using the `guard` combinator to allocate the reply channel.

```
fun swapEvt (SC ch, vOut) = guard (fn () => let
      val inCh = channel ()
      in
        choose [
            wrap (recvEvt ch, fn (vIn, outCh) => (send(outCh, vOut); vIn)),
            wrap (sendEvt (ch, (vOut, inCh)), fn () => recv inCh)
          ]
      end)
```

Two-server interaction using events

Server abstraction:

```
type server
val rpcEvt : server * req -> repl event
```

The client code is no longer a mess.

```
select [
     wrap (rpcEvt server1, fn repl1 => act1 repl1),
     wrap (rpcEvt server2, fn repl2 => act2 repl2)
  ]
```

Note that `select` is shorthand for `sync` ∘ `choose`.

Other abstractions

Events have been used to implement a wide range of abstractions in CML, including:

- Futures
- Promises (asynchronous RPC)
- Actors
- Join patterns

Other abstractions

Events have been used to implement a wide range of abstractions in CML, including:

- Futures
- Promises (asynchronous RPC)
- Actors
- Join patterns

Other abstractions

Events have been used to implement a wide range of abstractions in CML, including:

- Futures
- Promises (asynchronous RPC)
- Actors
- Join patterns

Other abstractions

Events have been used to implement a wide range of abstractions in CML, including:

- Futures
- Promises (asynchronous RPC)
- Actors
- Join patterns

Other abstractions

Events have been used to implement a wide range of abstractions in CML, including:

- Futures
- Promises (asynchronous RPC)
- Actors
- Join patterns

Example — distributed tuple spaces

The *Linda* family of languages use *tuple spaces* to organize distributed computation.
A tuple space is a shared associative memory, with three operations:

- **output** adds a tuple.
- **input** removes a tuple from the tuple space. The tuple is selected by matching against a *template*.
- **read** reads a tuple from the tuple space, without removing it.

```
val output : (ts * tuple) -> unit
val input  : (ts * template) -> value list event
val read   : (ts * template) -> value list event
```

Distributed tuple spaces *(continued ...)*

There are two ways to implement a distributed tuple space:

- *Read-all, write-one*
- *Read-one, write-all*

We choose read-all, write-one. In this organization, a `write` operation goes to a single processor, while an `input` or `read` operation must query all processors.

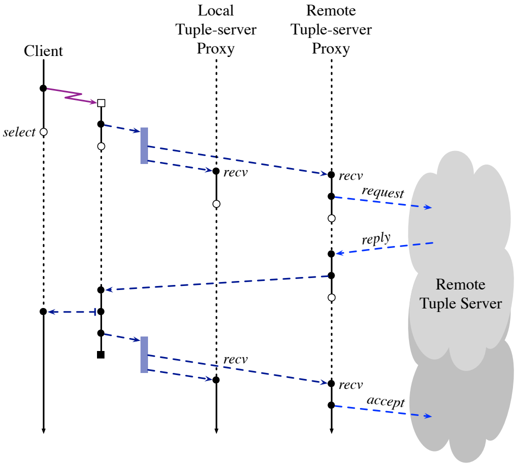
Distributed tuple spaces *(continued ...)*

The `input` protocol is complicated:

1. The reader broadcasts the query to all tuple-space servers.
2. Each server checks for a match; if it finds one, it places a hold on the tuple and sends it to the reader. Otherwise it remembers the request to check against subsequent `write` operations.
3. The reader waits for a matching tuple. When it receives a match, it sends an acknowledgement to the source, and cancellation messages to the others.
4. When a tuple server receives an acknowledgement, it removes the tuple; when it receives a cancellation it removes any hold or queued request.

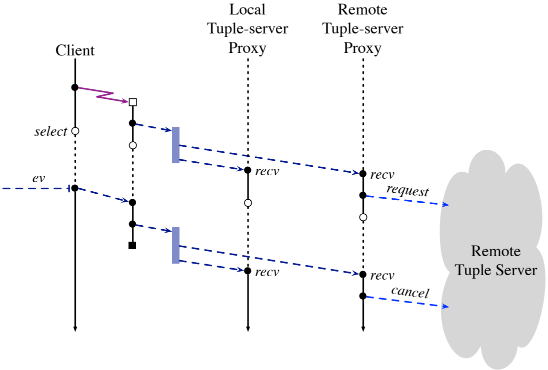
Distributed tuple spaces *(continued ...)*

Here is the message traffic for a successful `input` operation:



Distributed tuple spaces *(continued ...)*

We use negative acknowledgements to cancel requests when the client chooses some other event.



Note that we must confirm that a client accepts a tuple before sending out the acknowledgement.

Implementing concurrency in functional languages

- Functional languages can provide a platform for efficient implementations of concurrency features.
- This is especially true for languages that support continuations.

Implementing concurrency in functional languages

- Functional languages can provide a platform for efficient implementations of concurrency features.
- This is especially true for languages that support continuations.

Continuations

Continuations are a semantic concept that captures the meaning of the "rest of the program." In a functional language, we can apply the *continuation-passing-style* transformation to make continuations explicit.

For example, consider the expression "`(x+y)*z`." We can rewrite it as

```
(fn k => k(x+y)) (fn v => v*z)
```

In this rewritten code, the variable `k` is bound to the continiation of the expression "`x+y`."

First-class continuations

Some languages make it possible to reify the implicit continuations. For example, SML/NJ provides the following interface to its first-class continuations:

```
type 'a cont
val callcc : ('a cont -> 'a) -> 'a
val throw  : 'a cont -> 'a -> 'b
```

First-class continuations can be used to implement many kinds of control-flow, including loops, back-tracking, exceptions, and various concurrency mechanisms.

Coroutines

Implementing a simple coroutine package using continuations is straightforward.

```
val fork : (unit -> unit) -> unit
val exit : unit -> 'a
val yield : unit -> unit
```

Coroutines *(continued ...)*

```
val rdyQ : unit cont Q.queue = Q.mkQueue()

fun dispatch () = throw (Q.dequeue rdyQ) ()

fun yield () = callcc (fn k => (
      Q.enqueue (rdyQ, k);
      dispatch ()))

fun exit () = dispatch ()

fun fork f = callcc (fn parentK => (
      Q.enqueue (rdyQ, parentK);
      (f ()) handle _ => ();
      exit ()))
```

To support preemption and/or parallelism requires additional runtime-system support.