Concurrency and Parallelism in Functional Programming Languages

John Reppy
jhr@cs.uchicago.edu

University of Chicago

April 21 & 24, 2017
Outline

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

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

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

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

- Programming models
- Concurrent
- 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.
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.
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.
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.
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.
Parallel vs. concurrent vs. distributed

Parallel, concurrent, and distributed programming address different problems.

- Parallelism is about speed — exploiting parallel processors to solve problems quicker.
- Concurrency is about nondeterminism — managing the unpredictable external world.
- Distributed systems is about computing in a network — it involves aspects of both parallelism and concurrency, but also raises issues of fault-tolerance and trust.
Parallel vs. concurrent vs. distributed

Parallel, concurrent, and distributed programming address different problems.

- Parallelism is about speed — exploiting parallel processors to solve problems quicker.
- Concurrency is about nondeterminism — managing the unpredictable external world.
- Distributed systems is about computing in a network — it involves aspects of both parallelism and concurrency, but also raises issues of fault-tolerance and trust.
Parallel vs. concurrent vs. distributed

Parallel, concurrent, and distributed programming address different problems.

- **Parallelism** is about speed — exploiting parallel processors to solve problems quicker.
- **Concurrency** is about nondeterminism — managing the unpredictable external world.
- **Distributed systems** is about computing in a network — it involves aspects of both parallelism and concurrency, but also raises issues of fault-tolerance and trust.
Parallel vs. concurrent vs. distributed

Parallel, concurrent, and distributed programming address different problems.

- Parallelism is about speed — exploiting parallel processors to solve problems quicker.
- Concurrency is about nondeterminism — managing the unpredictable external world.
- Distributed systems is about computing in a network — it involves aspects of both parallelism and concurrency, but also raises issues of fault-tolerance and trust.
Implicitly parallel vs. implicitly threaded vs. explicitly threaded

In the space of parallel programming models, there are choices to be made about how programmers introduce parallelism.

- Implicitly parallel programming relies entirely on the compiler and runtime to determine when two computations should be run in parallel.
- Implicitly threaded parallelism relies on the programmer adding annotations that mark places where parallelism would be useful, but the language does not make explicit any notion of parallel threads.
- Explicit threading (aka concurrency) uses language-level threading mechanisms to specify parallelism.

Another, related, design axis is data-parallel vs. task-parallel
Implicitly parallel vs. implicitly threaded vs. explicitly threaded

In the space of parallel programming models, there are choices to be made about how programmers introduce parallelism.

- Implicitly parallel programming relies entirely on the compiler and runtime to determine when two computations should be run in parallel.
- Implicitly threaded parallelism relies on the programmer adding annotations that mark places where parallelism would be useful, but the language does not make explicit any notion of parallel threads.
- Explicit threading (aka concurrency) uses language-level threading mechanisms to specify parallelism.

Another, related, design axis is data-parallel vs. task-parallel
Implicitly parallel vs. implicitly threaded vs. explicitly threaded

In the space of parallel programming models, there are choices to be made about how programmers introduce parallelism.

- Implicitly parallel programming relies entirely on the compiler and runtime to determine when two computations should be run in parallel.
- Implicitly threaded parallelism relies on the programmer adding annotations that mark places where parallelism would be useful, but the language does not make explicit any notion of parallel threads.
- Explicit threading (aka concurrency) uses language-level threading mechanisms to specify parallelism.

Another, related, design axis is data-parallel vs. task-parallel
Implicitly parallel vs. implicitly threaded vs. explicitly threaded

In the space of parallel programming models, there are choices to be made about how programmers introduce parallelism.

- Implicitly parallel programming relies entirely on the compiler and runtime to determine when two computations should be run in parallel.
- Implicitly threaded parallelism relies on the programmer adding annotations that mark places where parallelism would be useful, but the language does not make explicit any notion of parallel threads.
- Explicit threading (aka concurrency) uses language-level threading mechanisms to specify parallelism.

Another, related, design axis is data-parallel vs. task-parallel
Implicitly parallel vs. implicitly threaded vs. explicitly threaded

In the space of parallel programming models, there are choices to be made about how programmers introduce parallelism.

- Implicitly parallel programming relies entirely on the compiler and runtime to determine when two computations should be run in parallel.
- Implicitly threaded parallelism relies on the programmer adding annotations that mark places where parallelism would be useful, but the language does not make explicit any notion of parallel threads.
- Explicit threading (aka concurrency) uses language-level threading mechanisms to specify parallelism.

Another, related, design axis is data-parallel vs. task-parallel
Deterministic vs. non-deterministic

Multiple threads/processors introduces the non-deterministic program execution; \textit{i.e.}, two runs of a program may produce different results.

Parallel languages that are implicitly parallel or implicitly threaded usually hide this non-determinism and guarantee \textit{sequential semantics}.

Explicitly threaded languages are naturally concurrent, although there are a few examples of deterministic concurrent languages.
Deterministic vs. non-deterministic

Multiple threads/processors introduces the non-deterministic program execution; *i.e.*, two runs of a program may produce different results.

Parallel languages that are implicitly parallel or implicitly threaded usually hide this non-determinism and guarantee sequential semantics.

Explicitly threaded languages are naturally concurrent, although there are a few examples of deterministic concurrent languages.
Deterministic vs. non-deterministic

Multiple threads/processors introduces the non-deterministic program execution; *i.e.*, two runs of a program may produce different results.

Parallel languages that are implicitly parallel or implicitly threaded usually hide this non-determinism and guarantee **sequential semantics**.

Explicitly threaded languages are naturally concurrent, although there are a few examples of deterministic concurrent languages.
Shared state vs. shared-nothing

The last design axis is sharing of state:

- Shared-memory uses the mechanisms of imperative programming to implement communication between threads.
- Shared-nothing requires that threads communicate via some form of messaging.

Note that shared-nothing languages can still be implemented in a shared address space!
Shared state vs. shared-nothing

The last design axis is sharing of state:

- Shared-memory uses the mechanisms of imperative programming to implement communication between threads.
- Shared-nothing requires that threads communicate via some form of messaging.

Note that shared-nothing languages can still be implemented in a shared address space!
Shared state vs. shared-nothing

The last design axis is sharing of state:

- Shared-memory uses the mechanisms of imperative programming to implement communication between threads.
- Shared-nothing requires that threads communicate via some form of messaging.

Note that shared-nothing languages can still be implemented in a shared address space!
Shared state vs. shared-nothing

The last design axis is sharing of state:

- Shared-memory uses the mechanisms of imperative programming to implement communication between threads.
- Shared-nothing requires that threads communicate via some form of messaging.

Note that shared-nothing languages can still be implemented in a shared address space!
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

There are two aspects to thread interaction:

- **Communication** — how does data get from one thread to another?
- **Synchronization** — how are the possible orderings of threads restricted?
  - Mutual-exclusion synchronization — protecting access to a shared resource
  - Condition synchronization — waiting for a signal from another thread

The choice of synchronization and communication mechanisms is a critical design choice

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

There are two aspects to thread interaction:

- **Communication** — how does data get from one thread to another?
- **Synchronization** — how are the possible orderings of threads restricted?
  - Mutual-exclusion synchronization — protecting access to a shared resource
  - Condition synchronization — waiting for a signal from another thread

The choice of synchronization and communication mechanisms is a critical design choice

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

There are two aspects to thread interaction:

- **Communication** — how does data get from one thread to another?
- **Synchronization** — how are the possible orderings of threads restricted?
  - **Mutual-exclusion synchronization** — protecting access to a shared resource
  - **Condition synchronization** — waiting for a signal from another thread

The choice of synchronization and communication mechanisms is a critical design choice

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

There are two aspects to thread interaction:

- **Communication** — how does data get from one thread to another?
- **Synchronization** — how are the possible orderings of threads restricted?
  - **Mutual-exclusion synchronization** — protecting access to a shared resource
  - **Condition synchronization** — waiting for a signal from another thread

The choice of synchronization and communication mechanisms is a critical design choice

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

There are two aspects to thread interaction:

- **Communication** — how does data get from one thread to another?
- **Synchronization** — how are the possible orderings of threads restricted?
  - **Mutual-exclusion synchronization** — protecting access to a shared resource
  - **Condition synchronization** — waiting for a signal from another thread

The choice of synchronization and communication mechanisms is a critical design choice

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

There are two aspects to thread interaction:

- **Communication** — how does data get from one thread to another?
- **Synchronization** — how are the possible orderings of threads restricted?
  - **Mutual-exclusion synchronization** — protecting access to a shared resource
  - **Condition synchronization** — waiting for a signal from another thread

The choice of synchronization and communication mechanisms is a critical design choice

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

There are two aspects to thread interaction:

- **Communication** — how does data get from one thread to another?
- **Synchronization** — how are the possible orderings of threads restricted?
  - **Mutual-exclusion synchronization** — protecting access to a shared resource
  - **Condition synchronization** — waiting for a signal from another thread

The choice of synchronization and communication mechanisms is a critical design choice

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

There are two aspects to thread interaction:

- **Communication** — how does data get from one thread to another?
- **Synchronization** — how are the possible orderings of threads restricted?
  - **Mutual-exclusion synchronization** — protecting access to a shared resource
  - **Condition synchronization** — waiting for a signal from another thread

The choice of synchronization and communication mechanisms is a critical design choice

- 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.

Classic example:

\[
x \leftarrow 0
\]

\[
parbegin x \leftarrow x + 1 \parallel x \leftarrow x + 1 \ parend
\]

\[
write x
\]
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.

Classic example:

```
x ← 0
parbegin x ← x + 1 || x ← x + 1 parend
write x
```
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.

Classic example:

\[
\begin{align*}
x & \leftarrow 0 \\
\text{parbegin } x & \leftarrow x + 1 \ || \ x \leftarrow x + 1 \ \text{parend} \\
\text{write } x
\end{align*}
\]
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.

Classic example:

```plaintext
x ← 0
parbegin x ← x + 1 || x ← x + 1 parend
write x
```
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.

Classic example:

```plaintext
x ← 0
parbegin x ← x + 1 || x ← x + 1 parend
write x
```
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.

Classic example:

\[
\begin{align*}
x & \leftarrow 0 \\
\text{parbegin} & \quad x \leftarrow x + 1 \mid x \leftarrow x + 1 \text{ parend} \\
& \quad \text{write } x
\end{align*}
\]
Software transactional memory

- Software transactional memory (STM) has been offered as a solution.
- Introduces atomic regions that are serialized with other atomic regions.

\[
\begin{align*}
x &\leftarrow 0 \\
\text{atomic} \begin{cases} 
  x &\leftarrow x + 1 \\
  x &\leftarrow x + 1
\end{cases} \quad || \quad \text{atomic} \begin{cases} 
  x &\leftarrow x + 1
\end{cases} \\
\text{write} \ x
\end{align*}
\]

- Uses non-blocking techniques to increase potential parallelism.
- Some hardware support in the latest processors.
- 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 not directly support conditional synchronization.
Software transactional memory

- Software transactional memory (STM) has been offered as a solution.
- Introduces atomic regions that are serialized with other atomic regions.

```plaintext
x ← 0
atomic { x ← x + 1 } || atomic { x ← x + 1 }
write x
```

- Uses non-blocking techniques to increase potential parallelism.
- Some hardware support in the latest processors.
- 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 not directly support conditional synchronization.
Software transactional memory

- Software transactional memory (STM) has been offered as a solution.
- Introduces **atomic** regions that are serialized with other atomic regions.

\[
x \leftarrow 0\\
\text{atomic} \{ x \leftarrow x + 1 \} \parallel \text{atomic} \{ x \leftarrow x + 1 \}\\
\text{write} \ x
\]

- Uses non-blocking techniques to increase potential parallelism.
- Some hardware support in the latest processors
- 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 not directly support conditional synchronization.
Software transactional memory

- Software transactional memory (STM) has been offered as a solution.
- Introduces **atomic** regions that are serialized with other atomic regions.

\[
x \leftarrow 0 \\
\text{atomic} \{ x \leftarrow x + 1 \} \ || \ \text{atomic} \{ x \leftarrow x + 1 \} \\
\text{write } x
\]

- Uses non-blocking techniques to increase potential parallelism.
- Some hardware support in the latest processors
- 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 not directly support conditional synchronization.
Software transactional memory

- Software transactional memory (STM) has been offered as a solution.
- Introduces **atomic** regions that are serialized with other atomic regions.

\[
x \leftarrow 0
\]

\[
\text{atomic} \{ x \leftarrow x + 1 \} \parallel \text{atomic} \{ x \leftarrow x + 1 \}
\]

\[
\text{write } x
\]

- Uses non-blocking techniques to increase potential parallelism.
- Some hardware support in the latest processors
  - 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 not directly support conditional synchronization.
Software transactional memory

- Software transactional memory (STM) has been offered as a solution.
- Introduces atomic regions that are serialized with other atomic regions.

\[
\begin{align*}
    x &\leftarrow 0 \\
    \text{atomic} \{ x \leftarrow x + 1 \} \ || \ \text{atomic} \{ x \leftarrow x + 1 \} \\
    &\text{write } x
\end{align*}
\]

- Uses non-blocking techniques to increase potential parallelism.
- Some hardware support in the latest processors
- 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 not directly support conditional synchronization.
Software transactional memory

- Software transactional memory (STM) has been offered as a solution.
- Introduces **atomic** regions that are serialized with other atomic regions.

```
x ← 0

atomic { x ← x + 1 } || atomic { x ← x + 1 }
write x
```

- Uses non-blocking techniques to increase potential parallelism.
- Some hardware support in the latest processors
- 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 not directly support conditional synchronization.
Software transactional memory

- Software transactional memory (STM) has been offered as a solution.
- Introduces **atomic** regions that are serialized with other atomic regions.

\[
x \leftarrow 0
\]
\[
\text{atomic}\ (x \leftarrow x + 1) \ || \ \text{atomic}\ (x \leftarrow x + 1)
\]
\[
\text{write } x
\]

- Uses non-blocking techniques to increase potential parallelism.
- Some hardware support in the latest processors
- 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 not directly support conditional synchronization.
**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.
- Natural fit for functional programming (threads are just tail-recursive functions).
- Inspired many language designs, including Concurrent ML, 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.
- Natural fit for functional programming (threads are just tail-recursive functions).
- Inspired many language designs, including Concurrent ML, go (and its predecessors), OCCAM, OCCAM-$\pi$, 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.
- Natural fit for functional programming (threads are just tail-recursive functions).
- Inspired many language designs, including Concurrent ML, go (and its predecessors), OCCAM, OCCAM-\(\pi\), etc..
Concurrency Message-passing

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.
- Natural fit for functional programming (threads are just tail-recursive functions).
- Inspired many language designs, including Concurrent ML, go (and its predecessors), OCCAM, OCCAM-$\pi$, 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.
- Natural fit for functional programming (threads are just tail-recursive functions).

- Inspired many language designs, including Concurrent ML, 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.
- Natural fit for functional programming (threads are just tail-recursive functions).
- Inspired many language designs, including Concurrent ML, go (and its predecessors), OCCAM, OCCAM-$\pi$, 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.
Concurrency

Message-passing

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

```ml
type 'a chan

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

We also need to define a way to create threads:

```ml
val spawn : (unit -> unit) -> unit
```
Example: concurrent streams

We can connect threads together with channels to implement concurrent streams.

Here is a function that creates the stream of integers 2, 3, 4, ...

```haskell
fun countFrom2 () = let
  val outCh = channel()
  fun lp n = (send(outCh, n); lp(n+1))
  in
    spawn (fn () => lp 2); outCh
  end
```

And here is a function that filters out multiples of a number from a stream

```haskell
fun filter (inCh, p) = let
  val outCh = channel()
  fun lp () = let
    val n = recv inCh
    in
      if (n mod p = 0) then lp() else (send(outCh, n); lp())
    end
  in
    spawn lp; outCh
  end
```
Example: concurrent streams *(continued ...)*

Using these two functions

```ml
val countFrom2 : unit -> int chan
val filter : int chan * int -> int chan
```

we can implement the Sieve of Eratosthenes for finding prime numbers:

```ml
fun sieve () = let
    val outCh = channel()
    fun head ch = let
        val p = recv ch
        in
            send (outCh, p);
            head (filter (ch, p))
        end
    in
        spawn (fn () => head (countFrom2 ()));
        outCh
    end
```
Example: client-server concurrency

The other common pattern in concurrent programming is client-server interactions. A very simple example is a memory cell with the following API:

```ocaml
type 'a cell

val cell : 'a -> 'a cell
val get : 'a cell -> 'a
val set : 'a cell * 'a -> unit
```

We define a datatype to represent the two kinds of client requests:

```ocaml
datatype 'a req = GET | SET of 'a
```

And we represent a cell by a pair of channels

```ocaml
datatype 'a cell = CELL of {
    reqCh : 'a req chan,
    replyCh : 'a chan
}
```
Example: client-server concurrency (continued ...)

The cell function creates a new server and returns the pair of channels used to communicate with it:

```ml
fun cell init = let
  val reqCh = channel() and replyCh = channel()
  fun lp state = (case (recv reqCh)
    of GET => (send(replyCh, state); lp state)
       | SET v => lp v)
  in
    spawn (fn () => lp init);
    CELL{reqCh = reqCh, replyCh = replyCh}
  end

We can then define the matching client-side operations

```ml
fun get (CELL{reqCh, replyCh}) = (send(reqCh, GET); recv replyCh)
fun set (CELL{reqCh, ...}, v) = send(reqCh, SET v)
```

Notice that the client and server message operations match; if they did not match, then there would be deadlock.
Choice

To support monitoring communications on multiple channels, we need a choice operator that allows a thread to block on multiple channels. For example, we might define the following function:

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

that takes a list of channels paired with actions and waits until a message is available on one of the channels.
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.

```ml
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

The conflict between choice and abstraction was the prime motivation behind the design of Concurrent ML.

- CML provides a uniform framework for synchronization: events.
- CML provides event combinators for constructing abstract protocols.
- Event provide a uniform framework for many different kinds of event constructors:
  - I-variables
  - M-variables
  - Mailboxes
  - Channels
  - Timeouts
  - Thread termination
  - Synchronous I/O
Concurrent ML

The conflict between choice and abstraction was the prime motivation behind the design of Concurrent ML.

- CML provides a uniform framework for synchronization: events.
- CML provides event combinators for constructing abstract protocols.
- Event provide a uniform framework for many different kinds of event constructors:
  - I-variables
  - M-variables
  - Mailboxes
  - Channels
  - Timeouts
  - Thread termination
  - Synchronous I/O
Concurrent ML

The conflict between choice and abstraction was the prime motivation behind the design of Concurrent ML.

- CML provides a uniform framework for synchronization: events.
- CML provides event combinators for constructing abstract protocols.
- Event provide a uniform framework for many different kinds of event constructors:
  - I-variables
  - M-variables
  - Mailboxes
  - Channels
  - Timeouts
  - Thread termination
  - Synchronous I/O
Concurrent ML

The conflict between choice and abstraction was the prime motivation behind the design of Concurrent ML.

- CML provides a uniform framework for synchronization: events.
- CML provides event combinators for constructing abstract protocols.
- Event provide a uniform framework for many different kinds of event constructors:
  - I-variables
  - M-variables
  - Mailboxes
  - Channels
  - Timeouts
  - Thread termination
  - Synchronous I/O
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.

```ml
type 'a event
```

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

```ml
val recvEvt : 'a chan -> 'a event
val sendEvt : 'a chan * 'a -> unit event
```
Events (continued ...)

Event operations:

- Event wrappers for post-synchronization actions:
  ```ml
  val wrap : ('a event * ('a -> 'b)) -> 'b event
  ```

- Event generators for pre-synchronization actions and cancellation:
  ```ml
  val guard : (unit -> 'a event) -> 'a event
  val withNack : (unit event -> 'a event) -> 'a event
  ```

- Choice for managing multiple communications:
  ```ml
  val choose : 'a event list -> 'a event
  ```

- Synchronization on an event value:
  ```ml
  val sync : 'a event -> 'a
  ```
Example: Swap channels

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

```ml
type 'a swap_chan

val swapChannel : unit -> 'a swap_chan
val swapEvt : 'a swap_chan * 'a -> 'a event
```
Example: Swap channels (continued ...)

The basic implementation of swap channels is straightforward.

```ml
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
```

The `select` function is shorthand for `sync o choose`. 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

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

by using the guard combinator to allocate the reply channel.

```ml
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:

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

The client code is no longer a mess.

```ml
select [
  wrap (rpcEvt server1, fn repl1 => act1 repl1 ),
  wrap (rpcEvt server2, fn repl2 => act2 repl2 )
]
```
Two-server interaction using events (continued ...)

The implementation of the server protocol is as before, but we can now package it up as an event-valued abstraction:

```ml
datatype server = SERVER of (req * repl chan * unit event) chan

fun rpcEvt (SERVER recCh, req) = withNack (fn nack => 
  let
    val replCh = channel ()
  in
    send (reqCh, (req, replCh, nack));
    revcEvt replCh
  end)
```

CML
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.

```ml
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 first-class 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 first-class 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 follows:

\[
(fn \ k => k(x+y)) \ (fn \ v => v*z)
\]

In this rewritten code, the variable \(k\) is bound to the continuation 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:

```plaintext
(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.

```ml
val fork : (unit -> unit) -> unit
val exit : unit -> 'a
val yield : unit -> unit
```
Coroutines (continued ...)

```ml
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 ()))
```
Adding synchronization

To allow our threads to communicate, we will add support for ivars, which are write-once synchronous variables.

```ocaml
type 'a ivar

fun ivar : unit -> 'a ivar
val get : 'a ivar -> 'a
val put : 'a ivar * 'a -> unit
```

An ivar can either be empty (possibly with waiting threads) or full with a value, as reflected in the following representation:

```ocaml
datatype 'a ivar_state
    = EMPTY of 'a cont list
    | FULL of 'a

datatype 'a ivar = IV of 'a ivar_state ref
```

Ivars are created in the empty state:

```ocaml
fun ivar = ref(EMPTY[[]])
```
Adding synchronization (continued ...)

To get a value from an ivar, we check its state and block if it is empty.

```plaintext
fun get (IV r) = (case !r
    of EMPTY waiting => callcc (fn resumeK => (r := EMPTY(resumeK :: waiting); dispatch()))
        | FULL v => v)

fun put (IV r, v) = (case !r
    of EMPTY waiting => (r := FULL v;
                             List.app (bindAndEnqueue v) waiting)
        | FULL v => raise Fail "already_set")
```

The tricky part is the `bindAndEnqueue` function, which turns an `'a cont` into a `unit cont` and then enqueues it on the scheduling queue.

```plaintext
fun bindAndEnqueue (v : 'a) (k : 'a cont) : unit = Q.enqueue (rdyQ,
    callcc (fn k' => (callcc (fn unitK => throw k' unitK);
                         throw k v)))
```
Preemption and parallelism

- We can add preemptive scheduling by representing timer interrupts as asynchronous operations that reify the program state as a continuation.
- Adding preemption does require a mechanism for masking interrupts.
- We can also extend this model to support multicore parallelism, but that requires low-level shared-memory synchronization mechanisms to prevent race conditions when accessing the scheduling queues, etc.
Preemption and parallelism

- We can add preemptive scheduling by representing timer interrupts as asynchronous operations that reify the program state as a continuation.
- Adding preemption does require a mechanism for masking interrupts.
- We can also extend this model to support multicore parallelism, but that requires low-level shared-memory synchronization mechanisms to prevent race conditions when accessing the scheduling queues, etc.
We can add preemptive scheduling by representing timer interrupts as asynchronous operations that reify the program state as a continuation.

Adding preemption does require a mechanism for masking interrupts.

We can also extend this model to support multicore parallelism, but that requires low-level shared-memory synchronization mechanisms to prevent race conditions when accessing the scheduling queues, etc.