Programming Language Theory

Concurrency and Message Passing
Message Passing

- Threads communicate via send and receive along channels
  - (instead of reads and writes of references)
  - (but can implement one on top of the other)

- **Synchronous** message passing
  - Block until matching senders and receivers communicate
  - Encode asynchronous sends by “spawn thread who sends (and blocks)”
Concurrent ML (CML)

Features:

- dynamic creation of threads and typed channels
  - very lightweight threads: time/space cost for thread creation is about the same as a function call
- rendezvous communication via synchronous message passing
- first-class synchronous operations, called events
  - wrap synchronization abstractions to create new ones
  - dynamically (i.e., at run time)
- automatic reclamation (garbage collection) of threads and channels
- pre-emptive scheduling of explicitly concurrent threads
- efficient implementation — both on uni- and multi-processors

- very elegant and under appreciated
Threads

Create a new independent flow of sequential control

$val\ spawn : (unit \rightarrow unit) \rightarrow tid$

... $spawn\ th$ ...

▶ $th$ is of type $unit \rightarrow unit$
▶ $spawn\ th$ is of type $tid$ (the type of a thread identifier)
▶ the thread that evaluates $spawn\ th$ is the $parent$
▶ the thread that evaluates $th\ ()$ is the $child$

Thread executes until the evaluation of its expression is complete
▶ an uncaught exception completes the evaluation

Threads are preemptively scheduled

Program executes until all threads have terminated or are blocked
Channels

By themselves, multiple concurrent threads are not very useful

Need mechanisms for communication and synchronization

Synchronous message passing on typed channels

```plaintext
type 'a chan
val channel : unit -> 'a chan
val recv : 'a chan -> 'a
val send : 'a chan * 'a -> unit
```
Channels

*Synchronous message passing* on typed channels

- a sender blocks until there is a matching receiver

```
Thread 1

send (c, 5)
```
Channels

*Synchronous message passing* on typed channels

- a sender blocks until there is a matching receiver

Thread 1

```plaintext
send (c, 5)
```

Thread 2

```plaintext
recv c
```
Channels

Synchronous message passing on typed channels

- a sender blocks until there is a matching receiver
Channels

*Synchronous message passing* on typed channels
- a receiver blocks until there is a matching sender

Thread 1
- `send (c, 5)`
- `continue ()`

Thread 2
- `recv c`
- `continue 5`
Channels

Synchronous message passing on typed channels:

- channels do not name the sender or receiver
- channels do not specify the direction of communication
- a channel may pass multiple values between multiple threads
- multiple threads may offer to \texttt{recv} or \texttt{send} on the same channel
- each \texttt{recv} is matched with exactly one \texttt{send}

- implementation needs collection of waiting senders \texttt{xor} receivers
Examples

Many examples

- Bank account with private reference
- Bank account with loop-carried state
- Updatable storage cells
- Stream of squares
- Sieve of Eratosthenes (stream of primes)
- Fibonacci Series
Example: Bank Account w/ private reference

datatype action = Put of real | Get of real

type acct = action channel * real channel

fun mkAcct () =
  let val inCh = channel ()
    val outCh = channel ()
    val bal = ref 0.0 (* state *)
  fun loop () =
    let val _ = case recv inCh of (* blocks *)
      Put f => bal := !bal + f
    | Get f => bal := !bal - f
    val _ = send (outCh, !bal) (* blocks *)
    in  loop ()
    end
    val _ = spawn loop (* launch "server" *)
  in  (inCh, outCh)
  end

fun get (inCh, outCh) f =
  (send inCh (Get f); recv outCh)

fun put (inCh, outCh) f =
  (send inCh (Put f); recv outCh)
Example: Bank Account w/ loop-carried state

datatype action = Put of real | Get of real
type acct = action channel * real channel
fun mkAcct () =
  let val inCh = channel ()
    val outCh = channel ()
    fun loop bal =
      let val _ = send (outCh, bal) (* blocks *)
      in loop bal
    end
    val _ = spawn (fn () => loop 0.0) (* launch "server" *)
  in (inCh, outCh)
end

fun get (inCh, outCh) f =
  (send inCh (Get f); recv outCh)

fun put (inCh, outCh) f =
  (send inCh (Put f); recv outCh)
Example: Bank Account

Note: can abstract all threading and communication away from clients:

type acct
val mkAcct : unit -> acct
val get : acct -> real -> real
val put : acct -> real -> real

Hidden thread communication:

- mkAcct spawns a thread (the “this account server”)
- get and put make the server go around the loop once

Races naturally avoided; the server handles one request at a time

- CML *implementation* has queues for pending communications
Example: Updatable Storage Cells

Although mutable state make concurrent programming difficult, it is easy to give an implementation of updatable storage cells using threads and channels.

Implementation is a prototypical example of the client-server style of concurrent programming.

signature CELL =
    sig
        type 'a cell
        val cell : 'a -> 'a cell
        val get : 'a cell -> 'a
        val put : 'a cell * 'a -> unit
    end
Example: Updatable Storage Cells

structure Cell : CELL =
  sig
    datatype 'a req = GET of 'a chan | PUT of 'a
    datatype 'a cell = CELL of 'a req chan
  
fun get (CELL reqCh) =
  let
    val replyCh = channel ()
  in
    send (reqCh, GET replyCh);
    recv replyCh
  end

fun put (CELL reqCh, y) =
  send (reqCh, PUT y)
fun cell z =
  let
    val reqCh = channel ()
    fun loop x =
      case recv reqCh of
        GET replyCh => (send (replyCh, x);
                          loop x)
      | PUT y => loop y
    val _ = spawn (fn () => loop z)
  in
    CELL reqCh
  end
end
Example: Stream of squares

Compute a stream of squares

fun mkSquares () =
  let val sqrCh = channel
    fun loop i =
      (send (sqrCh, i * i);
       loop (i + 1))
    val _ = spawn (fn () => loop 1)
    fun next () = recv sqrCh
  in  next
  end

(* val mkSquares : unit -> (unit -> int) *)
Example: Sieve of Eratosthenes

Compute a *stream* of prime numbers

Implementation is a prototypical example of the *dataflow* style of concurrent programming

```haskell
fun firstPrimes (n : int) : int list =
    let val primesCh = primes ()
    fun loop (i, acc) =
        if i = 0
            then rev acc
        else loop (i - 1, (recv primesCh)::acc)
    in  loop (n, [])
end
```
Example: Sieve of Eratosthenes

fun forever (init : 'a) (f : 'a -> 'a) : unit =
  let fun loop s = loop (f s)
    val _ = spawn (fn () => loop init)
  in ()
end

fun succs (i : int) : int chan =
  let val succsCh = channel ()
    fun succsFn i = (send (succsCh, i); i + 1)
    val () = forever i succsFn
  in succsCh
end
fun filter (p: int, inCh : int chan) : int chan = 
  let val outCh = channel ()
  fun filterFn () =
    let val i = recv inCh
    in  if (i mod p) <> 0 then send (outCh, i) else ()
    end
    val () = forever () filterFn
  in  outCh
  end

fun primes () : int chan =
  let val primesCh = channel ()
  fun primesFn ch =
    let val p = recvCh
    in  send (primesCh p) ; filter (p, ch)
    end
    val () = forever (sucss 2) primesFn
  in  primesCh
  end
Example: Fibonacci Series

Compute a *stream* of Fibonacci numbers

\[
\begin{align*}
  fib_1 &= 1 \\
  fib_2 &= 1 \\
  fib_{i+2} &= fib_{i+1} + fib_i
\end{align*}
\]
Example: Fibonacci Series

Compute a *stream* of Fibonacci numbers

\[
\begin{align*}
    fib_1 &= 1 \\
    fib_2 &= 1 \\
    fib_{i+2} &= fib_{i+1} + fib_i
\end{align*}
\]
Example: Fibonacci Series

fun addStrms (inCh1, inCh2, outCh) = 
  forever () (fn () =>
    send (outCh, (recv inCh1) + (recv inCh2)))

fun copyStrm (inCh, outCh1, outCh2) = 
  forever () (fn () =>
    let val x = recv inCh
    in  send (outCh1, x) ; send (outCh2, x)
    end)

fun delayStrm first (inCh, outCh) = 
  forever first (fn x =>
    (send (outCh, x) ; recv inCh))
fun fibs () : int chan =
  let val fibsCh = channel ()
  val ch1 = channel ()
  val ch2 = channel ()
  val ch3 = channel ()
  val ch4 = channel ()
  val ch5 = channel ()
  in
  copyStrm (ch1, ch2, fibsCh);
  copyStrm (ch2, ch3, ch4);
  delayStrm 0 (ch4, ch5);
  addStrms (ch3, ch5, ch1);
  send (ch1, 1);
  fibsCh
end
Need for Selective Communication

When programming with `recv` and `send` exclusively, there are limits to the kinds of concurrent programs that can be expressed.

- fragility in the implementation of concurrency abstractions
Need for Selective Communication

When programming with `recv` and `send` exclusively, there are limits to the kinds of concurrent programs that can be expressed.

- fragility in the implementation of concurrency abstractions
Need for Selective Communication

When programming with `recv` and `send` exclusively, there are limits to the kinds of concurrent programs that can be expressed.

- fragility in the implementation of concurrency abstractions

```
matlab
 % Example code
 % delay
 % copy
 % add
 % fibsCh
```

problem: deadlock

solution: eliminate dependency on the order of blocking operations
Need for Selective Communication

When programming with `recv` and `send` exclusively, there are limits to the kinds of concurrent programs that can be expressed.

- fragility in the implementation of concurrency abstractions

Matthew Fluet
Programming Language Theory
Lecture 19
Need for Selective Communication

When programming with `recv` and `send` exclusively, there are limits to the kinds of concurrent programs that can be expressed.

- fragility in the implementation of concurrency abstractions

- problem: *deadlock*

- solution: eliminate dependency on the order of blocking operations
Selective Communication

Selective communication

- allow a thread to block on a choice of several communications
- first communication that becomes enabled is chosen
- if two or more communications are simultaneously enabled, then one is chosen nondeterministically
Selective Communication vs. Abstraction

*Selective communication vs. Abstraction*

- in most concurrent languages with message passing, must explicitly list the blocking communications:

  ```
  select inCh1?x => x + (recv inCh2)
  | inCh2?y => (recv inCh1) + y
  | outCh!42 => 0
  ```

- makes it difficult to construct abstract synchronous operations, because constituent `recvs/?` and `sends/!` must be revealed, breaking abstraction
Selective Communication vs. Abstraction

Consider a possible interaction between a client and two servers.

Matthew Fluet
Programming Language Theory
Lecture 19
Selective Communication vs. Abstraction

Consider a possible interaction between a client and two servers

Without abstraction, the code is a mess:

```ml
let val replCh1 = channel ()
    val nackCh1 = channel ()
    val replCh2 = channel ()
    val nackCh2 = channel ()
in
    send (reqCh1, (req1, replyCh1, nackCh1));
    send (reqCh2, (req2, replyCh2, nackCh2));
    select replCh1?repl1 => (setNack nackCh2; act1 repl1)
    | replCh2?repl2 => (setNack nackCh1; act2 repl2)
end
```

Want an abstraction mechanism that supports choice
First-class Synchronous Operations

*First-class (abstract) synchronous operations (Events)*

- decouple the description of a synchronous operation from the act of synchronizing

*Events and synchronization*

- an event value represents a potential synchronous operations (analogy: a function value represents a potential computation)

```haskell
type 'a event
```

- force synchronization on an event value (analogy: application forces evaluation of a function value)

```haskell
val sync : 'a event -> 'a
```
First-class Synchronous Operations

First-class (abstract) synchronous operations (Events)

- decouple the description of a synchronous operation from the act of synchronizing

Base-event constructors

- event values that describe a primitive synchronous operation

- channel communication

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

  val recv = fn ch => sync (recvEvt ch)
  val send = fn (ch, x) => sync (sendEvt (ch, x))
Base-event Constructors for Channel Communication

Thread 1

sendEvt (c,5)

Thread 2

recvEvt c
Base-event Constructors for Channel Communication

Thread 1

sync

sendEvt (c,5)

Thread 2

recvEvt c
Base-event Constructors for Channel Communication

Thread 1
- sync
- sendEvt (c, 5)

Thread 2
- sync
- recvEvt c
Base-event Constructors for Channel Communication

Thread 1
- `sync`
- `sendEvt (c, 5)`
  - `continue ()`

Thread 2
- `sync`
- `recvEvt c`
  - `continue 5`
First-class Synchronous Operations

*First-class (abstract) synchronous operations (Events)*

- decouple the description of a synchronous operation from the act of synchronizing

*Event combinators*

- build more complicated event values from the base-event values
- generalized selective communication mechanism

```ocaml
val choose : 'a event * 'a event -> 'a event
```

- event wrapper for post-synchronization actions

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

- event generator for pre-synchronization actions

```ocaml
val guard : (unit -> 'a event) -> 'a event
```
Event Combinator for Generalized Choice

```plaintext
val choose : 'a event * 'a event -> 'a event
```

![Diagram showing the event combinator choose with two sendEvt expressions connected to it, one with (c2, 5) and the other with (c3, 5).]
Event Combinator for Generalized Choice

\[ \text{val choose : 'a event * 'a event -> 'a event} \]

Thread 1

\[ \text{sync} \]

\[ \text{choose} \]

\[ \text{sendEvt (c2,5)} \]

\[ \text{sendEvt (c3,5)} \]
Event Combinator for Generalized Choice

val choose : 'a event * 'a event -> 'a event

Thread 1

sync

choose

sendEvt (c2,5)  sendEvt (c3,5)
val choose : 'a event * 'a event -> 'a event
Event Combinator for Generalized Choice

val choose : 'a event * 'a event -> 'a event
val choose : 'a event * 'a event -> 'a event
Event Combinator for Post-synchronization Actions

val wrap : 'a event * ('a -> 'b) -> 'b event
Event Combinator for Post-synchronization Actions

\[
\text{val wrap : 'a event * ('a -> 'b) -> 'b event}
\]
val wrap : 'a event * ('a -> 'b) -> 'b event

Thread 1

sync

wrap f

recvEvt c
Event Combinator for Post-synchronization Actions

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

Thread 2

```
sync
sendEvt (c,5)
```

Thread 1

```
sync
wrap f
recvEvt c
```
val wrap : 'a event * ('a -> 'b) -> 'b event
Event Combinator for Post-synchronization Actions

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

```
Thread 1
sync
recvEvt c
wrap f
Thread 2
sync
sendEvt (c,5)
continue ()
```

```
Thread 1
sync
recvEvt c
wrap f

Thread 2
sync
sendEvt (c,5)
continue (f 5)
```
Using Event Combinators

fun addStrms (inCh1, inCh2, outCh) =  
  forever () (fn () =>  
    let val (a, b) =  
      sync (choose (  
        wrap (recvEvt inCh1, fn a => (a, recv inCh2)),  
        wrap (recvEvt inCh2, fn b => (recv inCh1, b))  
      ))  
    in  
      send (a + b)  
    end)

fun copyStrm (inCh, outCh1, outCh2) =  
  forever () (fn () =>  
    let val x = recv inCh  
    in  
      sync (choose (  
        wrap (sendEvt (outCh1, x), fn () => send (outCh2, x)),  
        wrap (sendEvt (outCh2, x), fn () => send (outCh1, x))  
      ))  
    end)
Event Combinator for Pre-synchronization Actions

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

fun f () =
  let val c = channel ()
    val _ = spawn (fn () => sync (sendEvt (c, 5)))
  in recvEvt c
  end
val guard : (unit -> 'a event) -> 'a event

fun f () =
  let val c = channel ()
  val _ = spawn (fn () => sync (sendEvt (c, 5)))
  in recvEvt c
  end
val guard : (unit -> 'a event) -> 'a event

fun f () =
  let val c = channel ()
  val _ = spawn (fn () => sync (sendEvt (c, 5)))
  in recvEvt c
  end
Event Combinator for Pre-synchronization Actions

\[
\text{val guard : (unit -> 'a event) -> 'a event}
\]

fun f () =
  let val c = channel ()
  val _ = spawn (fn () => sync (sendEvt (c, 5)))
  in recvEvt c
  end
Event Combinator for Pre-synchronization Actions

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

fun f () =
  let val c = channel ()
  val _ = spawn (fn () => sync (sendEvt (c, 5)))
  in recvEvt c
  end
**Event Combinator for Pre-synchronization Actions**

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

Thread 1
sync
recvEvt c
Thread 2
sync
sendEvt (c,5)
continue

fun f () =
  let val c = channel ()
  val _ = spawn (fn () => sync (sendEvt (c, 5)))
  in recvEvt c
  end
```
Example: Swap Channels

Swap Channels

- a synchronous abstraction
- allows (exactly) two threads to swap values

signature SWAP_CHAN =
  sig
    type 'a swap_chan
    val swapChannel : unit -> 'a swap_chan
    val swapEvt : 'a swap_chan * 'a -> 'a event
  end
Example: Swap Channels

structure BadSwapChan : SWAP_CHAN =
  struct
    datatype 'a swap_chan = SC of 'a chan

    fun swapChannel () = SC (channel ())

    fun swapEvt (SC ch, msgOut) =
      choose (
        wrap (recvEvt ch, fn msgIn =>
          (send (ch, msgOut); msgIn)),
        wrap (sendEvt (ch, msgOut), fn () =>
          recv ch)
      )
  end
Example: Swap Channels
Example: Swap Channels

structure SwapChan : SWAP_CHAN =
  struct
    datatype 'a swap_chan = SC of ('a * 'a chan) chan

    fun swapChannel () = SC (channel ())

    fun swapEvt (SC ch, msgOut) =
      guard (fn () =>
        let val inCh = channel ()
        in
          choose (            
            wrap (recvEvt ch, fn (msgIn, outCh) =>
              (send (outCh, msgOut) ; msgIn)),
            wrap (sendEvt (ch, (msgOut, inCh)), fn () =>
              recv inCh)
          )
        end)
      end)
  end
Additional First-class Synchronous Operations

*Base-event constructors*

- event values that describe a primitive synchronous operation

- base-event constructors for trivial synchronizations

```haskell
val alwaysEvt : 'a -> 'a event
val neverEvt : 'a event

val chooseList : 'a event list -> 'a event =
  fn l => foldl choose neverEvt l
```
### Additional First-class Synchronous Operations

**Event combinators**

- build more complicated event values from the base-event values

- event generator for pre-synchronization actions with cancellation

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

fun f nackEvt =
  let val _ = spawn (fn () => sync (choose (recvEvt c3, wrap (nackEvt, g))))
  in  sendEvt (c3, 5)
  end
val withNack : (unit event -> 'a event) -> 'a event

fun f nackEvt =  
    let val _ = spawn (fn () => sync (choose (recvEvt c3, wrap (nackEvt, g))))  
in    sendEvt (c3, 5) 
end
val withNack : (unit event -> 'a event) -> 'a event

fun f nackEvt =  
  let val _ = spawn (fn () => sync (choose (recvEvt c3, wrap (nackEvt, g))))  
in  sendEvt (c3, 5)  
end
val withNack : (unit event -> 'a event) -> 'a event

fun f nackEvt = 
  let val _ = spawn (fn () => sync (choose ( recvEvt c3, wrap (nackEvt, g))))
  in  sendEvt (c3, 5) 
  end
Event Combinator for Pre-sync Actions w/ Cancellation

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

fun f nackEvt =
  let val _ = spawn (fn () => sync (choose (recvEvt c3, wrap (nackEvt, g))))
  in  sendEvt (c3, 5)
end
Event Combinator for Pre-sync Actions w/ Cancellation

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

fun f nackEvt = 
  let val _ = spawn (fn () => sync (choose (recvEvt c3, wrap (nackEvt, g))))
in  sendEvt (c3, 5)
end
val withNack : (unit event -> 'a event) -> 'a event

fun f nackEvt =  
  let val _ = spawn (fn () => sync (choose (recvEvt c3, wrap (nackEvt3, g))))  
in  sendEvt (c3, 5)  
end
val withNack : (unit event -> 'a event) -> 'a event

fun f nackEvt = 
  let val _ = spawn (fn () => sync (choose (recvEvt c3, wrap (nackEvt3, g))))
  in sendEvt (c3, 5)
  end
Selective Communication vs. Abstraction

Consider a possible interaction between a client and two servers.

Without abstraction, the code is a mess:

```ocaml
let val replCh1 = channel ()
val nackCh1 = channel ()
val replCh2 = channel ()
val nackCh2 = channel ()

in
  send (reqCh1, (req1, replyCh1, nackCh1));
  send (reqCh2, (req2, replyCh2, nackCh2));
  select replCh1?repl1 => (setNack nackCh2 ;
                              act1 repl1)
  | replCh2?repl2 => (setNack nackCh1 ;
                      act2 repl2)
end
```

Matthew Fluet
Programming Language Theory
Lecture 19
Selective Communication vs. Abstraction

Consider a possible interaction between a client and two servers

With abstraction, the code is clean:

```ocaml
structure Server : sig
  val rpcEvt : server * req -> repl event
end = struct
  fun rpcEvt (srv, req) =
    withNack (fn nack =>
      let val replyCh = channel
      in
        ... send (reqCh, (req, replyCh, nack)) ... ;
        recvEvt replyCh
      end)
end

sync (choose ( wrap (Server.rpcEvt server1, fn repl1 => act1 repl1),
               wrap (Server.rpcEvt server2, fn repl2 => act2 repl2) ))
```
External Synchronous Events

Motivations for concurrent programming:
▶ application domains with naturally concurrent structure:
   ▶ interactive systems (e.g., graphical-user interfaces)

Interactive systems
▶ multiple (asynchronous) input streams
   ▶ keyboard, mouse, network
▶ multiple (asynchronous) output streams
   ▶ display, audio, network

In sequential languages, dealt with through complex event loops and callback functions

First-class synchronous events can treat these external events using the same framework as internal synchronization
External Synchronous Events: Input/Output

For a console application, take standard input, output, and error streams to be character channels

```ocaml
val stdInCh : char chan
val stdOutCh : char chan
val stdErrCh : char chan
```

Better interface is to expose the streams as events

- should only `recv` from standard input stream
- should only `send` to standard output and error streams

```ocaml
val stdInEvt : char event
val stdOutEvt : char -> unit event
val stdErrEvt : char -> unit event
```

In practice, build higher-level I/O library on top
External Synchronous Events: Timeouts

Mechanisms for “timing out” on a blocking operation

```ml
val timeOutEvt : time -> unit event
val atTimeEvt : time -> unit event
```

Pause for one second

```ml
sync (timeOutEvt (timeFromSeconds 1))
```

Prompt for Y/N with default

```ml
choose (
  wrap (timeOutEvt (timeFromSeconds 10), fn () => #"N"),
  stdInEvt
)
```
Examples

Two final examples

- Buffered channels

- Futures
Example: Buffered Channels

Sometimes useful to support asynchronous communication

▶ sender does not block, message is buffered in the channel
▶ receiver blocks until there is an available message

```
signature BUFFERED_CHAN =
  sig
    type 'a buff-chan
    val buffChannel : unit -> 'a buff-chan
    val buffSend : 'a buff-chan * 'a -> unit
    val buffRecvEvt : 'a buff-chan -> 'a event
  end
```
Example: Buffered Channels

structure BufferedChan : BUFFERED_CHAN =
  struct
    datatype 'a buff_chan =
      BC of {inCh: 'a chan, outCh: 'a chan}

    fun buffSend (BC {outCh, ...}, x) =
      send (outCh, x)

    fun buffRecvEvt (BC {inCh, ...}) =
      recvEvt inCh
fun buffChannel () : 'a buff_chan =
  let val (inCh, outCh) = (channel (), channel ())
  fun loop ([], []) = loop ([recv inCh], [])
  | loop ([], rear) = loop (rev rear, [])
  | loop (front as frHd::frTl, rear) =
    (loop o sync o choose) (\(wrap (recvEvt inCh, \(fn \ y =>\)
      (front, \ y::\rear)),
      \(wrap (sendEvt (outCh, frHd), \(fn () =>\)
      (frTl, \rear)))
    )
  val _ = spawn (fn () => loop ([], []))
in  BC {inCh = inCh, outCh = outCh}
end
end
Example: Futures

*Futures*: a common mechanism for specifying parallel computation

- future creation: takes a computation, creates a separate thread and returns a placeholder (*future cell*)
- future touching: read a value from a future cell, blocking until value is computed

```
signature FUTURE =
  sig
    datatype 'a result = VAL of 'a | EXN of exn
    val future : ('a -> 'b) -> 'a -> 'b result event
  end
```
Example: Futures

structure Future : FUTURE =
  struct
    datatype 'a result = VAL of 'a | EXN of exn
fun future f x =
  let val ch = channel ()
  let val _ = spawn (fn () =>
    let val r = (VAL (f x)) handle exn => EXN exn
    in forever () (fn () => (send (ch, r)))
    end)
  in recvEvt ch
  end
Limitations

CML is (by design) for point-to-point communication

- Provably impossible to do things like 3-way swap
  (without busy-waiting or higher-level protocols or . . .)
- Provably impossible to do things like guarded receive
  (without busy-waiting or higher-level protocols or . . .)
- Related to issues of common-knowledge, especially in a
distributed-systems setting

Transactional Events:

- first-class synchronous message-passing events
- combined with atomic transactions
- gives the computation structure of a Monad

- one of my research projects, joint w/ Kevin Donnelly
- come talk to me to find out more
A note on implementations

CML encourages using \textit{lots} (100,000s) of threads

- Example: X Window library with one thread per widget

Threads should be cheap to support this paradigm

- SML/NJ: about as expensive as making a closure!
  - Think “current stack” plus a few words
  - Cost no time when blocked on a channel (dormant)
- MLton: \(\approx\) an order of magnitude \((\text{time})\) more expensive than a closure!
  - Could be cheaper; might be an interesting project
- OCaml: Similarly, not as cheap as SML/NJ’s