Message Passing

- Threads communicate via send and receive along channels instead of read and write of references
- Not so different? (can implement references on top of channels and channels on top of references)
- Synchronous message-passing
  - Block until communication takes place
  - Encode asynchronous by “spawn someone who blocks”

Concurrent ML

- CML is synchronous message-passing with first-class synchronization events
  - Can wrap synchronization abstractions to make new ones
  - At run-time
- Originally done for ML and fits well with lambdas, type-system, and implementation techniques, but more widely applicable
  - Available in Racket, OCaml, Haskell, ...
- Very elegant and under-appreciated
- Think of threads as very lightweight
  - Creation/space cost about like a function call

The Basics

```ocaml
type 'a channel (* messages passed on channels *) val new_channel : unit -> 'a channel

type 'a event (* when sync'ed on, get an 'a *) val send : 'a channel -> 'a -> unit event val receive : 'a channel -> 'a event
val sync : 'a event -> 'a
```

- Send and receive return “events” immediately
- Sync blocks until “the event happens”
- Separating these is key in a few slides
Simple version

Can define helper functions by trivial composition:

```ocaml
let sendNow ch a = sync (send ch a) (* block *)
let recvNow ch = sync (receive ch) (* block *)
```

“Who communicates” is up to the CML implementation
  ▶ Can be nondeterministic when there are multiple senders/receivers on the same channel
  ▶ Implementation needs collection of waiting senders xor receivers

Terminology note:
  ▶ Function names are those in OCaml’s Event library.
  ▶ In SML, the CML book, etc.:
    send ⇝ sendEvt
    receive ⇝ recvEvt
    sendNow ⇝ send
    recvNow ⇝ recv

Bank Account Example

See lec21code.ml

  ▶ First version: In/out channels are only access to private reference
    ▶ In channel of type action channel
    ▶ Out channel of type float channel
  ▶ Second version: Makes functional programmers smile
    ▶ State can be argument to a recursive function
    ▶ “Loop-carried”
    ▶ Hints at deep connection between references and channels
      ▶ Can implement the reference abstraction in CML

The Interface

The real point of the example is that you can abstract all the threading and communication away from clients:

```ocaml
type acct
val mkAcct : unit -> acct
val get : acct -> float -> float
val put : acct -> float -> float
```

Hidden thread communication:
  ▶ mkAcct makes 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 waiting communications

Streams

Another pattern/concept easy to code up in CML is a stream
  ▶ An infinite sequence of values, produced lazily (“on demand”)

Example in lec21code.ml: square numbers

Standard more complicated example: A network of streams for producing prime numbers. One approach:
  ▶ First stream generates 2, 3, 4, ...
  ▶ When the last stream generates a number \( p \), return it and dynamically add a stream as the new last stream
    ▶ Draws input from old last stream but outputs only those that are not divisible by \( p \)

Streams also:
  ▶ Have deep connections to circuits
  ▶ Are easy to code up in lazy languages like Haskell
  ▶ Are a key abstraction in real-time data processing
Wanting choice

▶ So far just used sendNow and recvNow, hidden behind simple interfaces

▶ But these block until the rendezvous, which is insufficient for many important communication patterns

▶ Example: add : int channel -> int channel -> int
  ▶ Must choose which to receive first; hurting performance if other provider ready earlier

▶ Example: or : bool channel -> bool channel -> bool
  ▶ Cannot short-circuit

This is why we split out sync and have other primitives

Choose and Wrap

type 'a event (* when sync’ed on, get an 'a *)
val send : 'a channel -> 'a -> unit event
val receive : 'a channel -> 'a event
val sync : 'a event -> 'a

val choose : 'a event list -> 'a event
val wrap : 'a event -> ('a -> 'b) -> 'b event

▶ choose: when synchronized on, block until one of the events happen (cf. UNIX select, but more useful to have sync separate)
▶ wrap: an event with the function as post-processing
  ▶ Can wrap as many times as you want

Note: Skipping a couple other key primitives (e.g., withNack for timeouts)

Circuits

To an electrical engineer:
▶ send and receive are ends of a gate
▶ wrap is combinational logic connected to a gate
▶ choose is a multiplexer
▶ sync is getting a result out

To a programming-language person:
▶ Build up a data structure describing a communication protocol
▶ Make it a first-class value that can be by passed to sync
▶ Provide events in interfaces so other libraries can compose larger abstractions

What can’t you do

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)
▶ Related to issues of common-knowledge, especially in a distributed setting
▶ Metamoral: Being a broad computer scientist is really useful
A note on implementation and paradigms

CML encourages using lots (100,000s) of threads

- Example: X Window library with one thread per widget

Threads should be cheap to support this paradigm

- SML N/J: about as expensive as making a closure!
  - Think “current stack” plus a few words
  - Cost no time when blocked on a channel (dormant)
- OCaml: Not cheap, unfortunately

A thread responding to channels is a lot like an asynchronous object (cf. actors)