Lec 26 Memoisation with parallelism

Things that can show up in 15-418, 15-312, 15-410

# Sequential impl

```
fun memoise f =
  let
    val cache = ref Table.empty
    fun g a = case△ Table.find !cache a of
                 SOME r => r
               ▷| NONE => let
                   val r = f g a ]← multi threads can call f if they have
                                    same a and a not in table
                   val _ = cache := Table.insert cache (a, r)
                 in◁             □   ↑
                                 two threads can insert
                   r             at same time
                 end
  in
    g
  end
```

<u>Prob</u> we often do ≥ 2 recursive calls and want to do them in parallel, but this impl not safe for parallelism

<u>Idea</u> suspend execution, make sure to not race compute

- at △, insert busy marker, at □, update actual result
  so lookup result can be busy, some, none.
- at ▷, handle busy case by
  - busy wait
  - sleep wait (OS could schedule some other work ?)
  - suspend job (SML built-in, but not Python / C++, etc.)
    give continuation / handle to another thread
    └ put self in some queue (could just be set)
      try wake things up at ◁
    SML: callcc to suspend
         throw to wake up

<u>Impl</u> initialise empty queue Q
states for table entry state = wait of Q | full of β

```
fun g a = case Table.find !cache a of
   NONE => insert (a, wait (empty queue)) to table ;
           r = f g a ;
```

```
            get queue Q ;
            insert (a, full r) ;
            wake up everything in Q ] ← O(1) span with some impl
                                        that supports parallel map
      | SOME (wait Q)  ⇒ suspend self in Q
      | SOME (full r)  ⇒ r
```

Also, make sure table and queue are linearisable
concurrent data struct needed.

Concurrent table

```
insert :  ctable → (α × β) → β option
            T       (a , b )
          if a not in T, add (a,b) and return NONE
          if (a, b') in T, return SOME b'  ← can retry with update
                                              in this case

update :  ctable → (α, β) → ()

let
  cache = Table.empty
  fun g a =
    let
      Q = Queue.empty
    in
      case (CTable.insert cache (a, wait Q)) of
        SOME (full r) ⇒ r
      | SOME (wait Q) ⇒
          suspend self onto Q ;
          when wake up, find result and return
      | NONE ⇒
             ⋮ (as before)
```