Concurrent Programming

Bernard Sufrin

Hilary Term 2016-17

10: CSO vs GO

Go – the language

* Designers say "Based on CSP" – meaning: "based on **occam**"
* Designers say "Tony Hoare ... genius ..."
* Designers have good reason to be opinionated – which doesn't make all their choices correct
* Nevertheless despite its warts and idiosyncrasies
  - The language is acceptably high-level (just!)
  - The language is well-implemented:
    * concurrency scales well: even to tens of millions of (small) running processes
    * context switch overhead is generally very low
    * thread resources used on host can be controlled
    * compilation and linking is fast
    * rich library / simple packaging scheme / community engagement encouraged
* It occupies a different ecological niche to Scala/CSO and Scala/Actors
* It rewards the careful programmer who can keep multiple details in mind at once

Integration in Go

```
type double float64
type funtype func (double) double

// Specification of an integration task: f, a, b, strips, (b-a)/strips
type task struct {f funtype; a double; b double; strips int; δ double}

func integral(f funtype, a double, b double, strips int, δ double) double {
     var sum double = (f(a) + f(b)) / 2.0
     var x = a
     for i:=1; i<strips; i++ { x = x + δ; sum = sum + f(x) }
     return sum * δ
}

func sq(x double) double { return x*x }
```

* Typed channels (buffered and synchronous)
* Typed Ports
* *Pointers*
* *Declarations with inferred types*

```
// Read a task; do the calculation; send the result back to the farmer
func workOnce(fromFarmer <-chan *task, toFarmer chan<- double) {

     // declare task to be a variable of type *task
     // initialised ( task := ) by reading ( <-fromFarmer ) from the input port
     task   := <-fromFarmer

     // declare result to be a variable of type double
     // initialised ( result := ) by invoking integral

     result := integral(task.f, task.a, task.b, task.strips, task.δ)

     // write the result to the toFarmer port
     toFarmer<- result
}
```

* Channels can be closed **only at their output end**
* The (overloaded) `range` notation iterates (reading) over an open channel

```
// workMany -- repeatedly read a task; do the calculation; send the result back to the farmer
func workMany(fromFarmer <-chan *task, toFarmer chan<- double ) {

     // repeatedly read a task from the input port
     for task   := range(fromFarmer) {

         // compute the integral, assigning it to a newly-declared variable
         result := integral(task.f, task.a, task.b, task.strips, task.δ)

         // and write it to the output port
         toFarmer <- result
     }
}
```

* **go** *function*(*arguments*) – forks a new process
* **make** – makes new channels

```
func manyTrapezium(f funtype, a double, b double, strips int, ntasks int, nworkers int) double {

    // Declare the three channels
    toWorkers      := make(chan *task)
    fromWorkers    := make(chan double)
    fromController := make(chan double)

    // Start all the worker processes
    for i:=0; i<nworkers; i++ { go workMany (toWorkers, fromWorkers) }

    // start the controller process
    go manyController(f, a, b, strips, ntasks, toWorkers, fromWorkers, fromController)

    // Await the result from the controller process; and return its value
    return <-fromController
}
```

```
func manyController(f funtype, a double, b double, strips int, ntasks int,
                    toWorkers chan<- *task, fromWorkers <-chan double,  toSystem chan<- double) () {

     δ          := (b-a)/(double(strips))
     taskSize   := strips / ntasks
     taskWidth  := (b - a) /double(ntasks)

     distributor :=     // construct and send ntasks task records to the workers
     func () {
          left := a
          for i:=0; i<ntasks; i++ {
              right := left+taskWidth
              toWorkers<-&task{f, left, right, taskSize, δ}
              left = right
          }
          close(toWorkers)
     }

     collector :=       // collect and sum ntasks results, and send the result to the system
     func () {
          result := double(0)
          for i:=0; i<ntasks; i++ { result = result + <- fromWorkers }
          toSystem <- result
     }

     PAR(distributor, collector).RUN() // run distributor and collector in parallel
}
```

* Functions are first-class objects
* **PAR** is not primitive
* `&task(...)` constructs a new task record and returns a pointer to it

Implementation of PAR (Sufrin): first approximation

```
// A PROCess is a statement abstracted as a func()()
type PROC  func()()

// SKIP is the unit of PAR: SKIP.PAR(p) ≡ p ≡ p.PAR(SKIP)
func SKIP() {}

// p.RUN() ≡ p()
func (p PROC) RUN() { p() }

// The PAR function takes several PROCs and returns a PROC that (when run) runs them concurrently
// (it is implemented here by a fold that uses the PAR method)
func PAR(procs ... PROC) (PROC) {
  if len(procs)==0 return SKIP
  result := procs[0]
  for i:=1; i<len(procs); i++ { result = result.PAR(procs[i]) }
  return result
}
```

* **Named** types can be associated with methods
* Above we defined RUN as a PROC method
* Below we will define PAR as a PROC method

```
// A status records the identity of a terminating component of a PAR
type status struct { id int }

func (l PROC) PAR (r PROC) PROC {
     return func () {
       sync := make(chan *status, 2) // buffered
       // Evaluate a PROC with a given integer identity and report its termination status to sync
       run  := func (id int, proc PROC) {
                    // defer means ‘‘call this function at termination or at panic (an exception was thrown)’’
                    defer func(){
                       // recover() yields nil or the value with which panic (throw an exception) was called
                       err:=recover()
                       // write the status of the terminating process to sync
                       if err!=nil { sync<- &status{id} } else { sync<- nil }
                    } ()
               // start the procedure
               proc()
             }

      // fork two processes
      go run (0, l)
      go run (1, r)

      // reap the statuses: recording if anything failed
      failed := false
      for j:=0; j<2; j++ { status := <-sync; failed = failed || status!=nil }

      // propagate any failure
      if failed { panic(PARERROR) }
     }
}
```

In the real implementation we record the detail of what failed and why

Forking Hell!

```
// A HANDLE represents a FORKed PROC that is currently running
type HANDLE struct { termination chan interface{} }

//
// proc.FORK() runs proc in a fresh GoRo and returns a handle, h, such that  h.WAIT()
// blocks until the running GoRo terminates (normally or by calling panic(...))
func (proc PROC) FORK() (HANDLE) {
     sync := HANDLE{make(chan interface{})}
     go func () {
           defer func(){ sync.termination<-recover() } ()
           proc()
        } ()
     return sync
}


// h.WAIT() waits for termination of the running GoRo with handle h
// and returns err if it terminated with panic(err), and nil otherwise
func (h HANDLE) WAIT() (interface{}) {
     // Reads the status from the termination channel: ok is false if the channel was closed
     err, ok  := <- h.termination
     if ok { close(h.termination) } else { panic(fmt.Sprintf("WAIT twice on HANDLE %v", h)) }
     return err
}
```

SELECT has a wart

* The analogue to ALT (namely SELECT) does not support boolean guards
* This can lead to some horrible ad-hoc programming and/or ad-hoc hacks, for example:

```
func farmer(a[] int, jobin <-chan JOB, jobout chan<- JOB) (PROC) {
     return func() {
       q        := newJOBS(2*M)
       working  := 0
       q.push(JOB{0, len(a)})

       // There is no WHILE construct: FOR plays multiple roles
       for working > 0 || !q.isEmpty() {
         jobsin  := jobin
         jobsout := jobout

         // this is the closest we get to ALT type guards
         if q.isEmpty() { jobsout = nil } // disable job output if no jobs waiting
         if working==0  { jobsin  = nil } // disable job input if no workers working

         // q.peek is evaluated every time the select is entered;
         // even though the output will not happen if jobsout is disabled
         // so q.peek() must not panic, even if q.isEmpty
         select { case jobsout<-q.peek(): q.pop()
                  case job := <- jobsin:  if job.l < 0 { working-- } else { q.push(job) }
                }
       }
       close(jobout)
     }
}
```

Contents

Note 1: 1 ☞
Meaning: "We curtseyed to Hoare: this gives us a licence to do things our own way if Hoare's advice proves inconvenient"