Horac

Ерин Игорь Антонович, 21.Б10

Санкт-Петербург
2022

Писать параллельные программы сложно

- Трудно воспроизводимые ошибки
  - Deadlocks
  - Race conditions
- Работа с низкоуровневыми сущностями
  - Потоки
  - Примитивы синхронизации

Concurrent ML

- Concurrent ML
  - Hopac
  - Racket
  - Clojure
  - Go

Основные сущности

- Потоки (Job)
- Каналы (Ch)
  - First-class
  - Higher-orde
  - Selective
  - Synchronous
  - Lightweight
- Альтернаантивы (Alt)

Hello world

```
let helloWorldJob = job {
  printfn "Hello, World!"
}
```

Updatable storage cell interface

```
type Cell<'a>
val cell: 'a -> Job<Cell<'a>>
val get: Cell<'a> -> Job<'a>
val put: Cell<'a> -> 'a -> Job<unit>
```

Updatable storage cell request

```
type Request<'a> =
  | Get
  | Put of 'a
```

Updatable storage cell

```
type Cell<'a> = {
  reqCh: Ch<Request<'a>>
  replyCh: Ch<'a>
}

let put (c: Cell<'a>) (x: 'a) : Job<unit> = job {
  return! Ch.give c.reqCh (Put x)
}

let get (c: Cell<'a>) : Job<'a> = job {
  do! Ch.give c.reqCh Get
  return! Ch.take c.replyCh
}
```

Cell constructor

```
let cell (x: 'a) : Job<Cell<'a>> = job {
  let c = {reqCh = Ch (); replyCh = Ch ()}
  let rec server x = job {
        let! req = Ch.take c.reqCh
        match req with
          | Get ->
            do! Ch.give c.replyCh x
            return! server x
          | Put x ->
            return! server x
      }
  do! Job.start (server x)
  return c
}
```

Cell example

```
> let c = run (cell 1) ;;
val c : Cell<int> = ...
> run (get c) ;;
val it : int = 1
> run (put c 2) ;;
val it : unit = ()
> run (get c) ;;
val it : int = 2
```

Garbage Collection

```
> GC.GetTotalMemory true ;;
val it : int64 = 39784152L
> let cs = ref (List.init 100000 <| fun i -> run (cell i)) ;;
// ...
> GC.GetTotalMemory true ;;
val it : int64 = 66296336L
> cs := [] ;;
val it : unit = ()
> GC.GetTotalMemory true ;;
val it : int64 = 39950064L
```

Combinators

```
let put (c: Cell<'a>) (x: 'a) : Job<unit> =
  Ch.give c.reqCh (Put x)

let get (c: Cell<'a>) : Job<'a> =
  Ch.give c.reqCh Get >>=. Ch.take c.replyCh

let create (x: 'a) : Job<Cell<'a>> = Job.delay <| fun () ->
  let c = {reqCh = Ch (); replyCh = Ch ()}
  let rec server x =
    Ch.take c.reqCh >>= function
      | Get ->
        Ch.give c.replyCh x >>=. server x
      | Put x -> server x
  Job.start (server x) >>-. c
```

Combinators

```
let put c x = c.reqCh *<- Put x

let get c = c.reqCh *<- Get >>=. c.replyCh

let create x = Job.delay <| fun () ->
  let c = {reqCh = Ch (); replyCh = Ch ()}
  Job.iterateServer x <| fun x ->
        c.reqCh >>= function
          | Get -> c.replyCh *<- x >>-. x
          | Put x -> Job.result x
  >>-. c
```

Updatable storage cell interface

```
type Cell<'a>
val cell: 'a -> Job<Cell<'a>>
val get: Cell<'a> -> Job<'a>
val put: Cell<'a> -> 'a -> Job<unit>
```

Updatable storage cell

```
type Cell<'a> = {
  getCh: Ch<'a>
  putCh: Ch<'a>
}

let get (c: Cell<'a>) : Job<'a> = Ch.take c.getCh

let put (c: Cell<'a>) (x: 'a) : Job<unit> = Ch.give c.putCh x
```

Selective communication

```
type Cell<'a> = {
  getCh: Ch<'a>
  putCh: Ch<'a>
}

let cell x = Job.delay <| fun () ->
  let c = { getCh = Ch (); putCh = Ch () }

  let rec server x =
    Alt.choose [ Ch.take c.putCh    ^=> fun x -> server x
                 Ch.give c.getCh x ^=> fun () -> server x ]
  Job.start (server x) >>-. c
```

Combinators

```
type Cell<'a> = {
  getCh: Ch<'a>
  putCh: Ch<'a>
}

let cell x = Job.delay <| fun () ->
  let c = { getCh = Ch (); putCh = Ch () }

  Job.server << Job.iterate x <| fun x ->
        Alt.choose [ Ch.take c.putCh
                     Ch.give c.getCh x ^->. x ]
  >>-. c
```

Naive fibonacci

```
let (<&>) xJ yJ =
  xJ >>= fun x -> yJ >>= fun y -> result (x, y)

let (>>-) xJ x2y =
  xJ >>= fun x -> result (x2y x)

let rec fib n = Job.delay <| fun () ->
  if n < 2L then
    Job.result n
  else
    fib (n-2L) <&> fib (n-1L) >>- fun (x, y) ->
    x + y
```

Parallel fibonacci

```
let rec fib n = Job.delay <| fun () ->
  if n < 2L then
    Job.result n
  else
    fib (n-2L) <*> fib (n-1L) >>- fun (x, y) ->
    x + y
```

Deadlock

```
let notSafe = Job.delay <| fun () ->
  let c = Ch ()
  Ch.take c <*> Ch.give c ()
```