Nondeterministic parallel computation in Pen

August 7th, 2022

@raviqqe

Overview of Pen

- Functional programming
- Immutable values
- Inductive values
  - Reference counting with in-place mutation
  - No circular dependency
- Capability-based effect system
  - Pass down "effect arguments" to functions.
- Parallel computation without data race
  - Synchronization by data structures (e.g. thunks, lazy lists, etc.)

Examples

Capability-based effect system

```
import Os'Console

main = \(os Os) none | error {
  Console'Print(os, "Hello, world!")?

  none
}
```

Nondeterministic parallel computation

- Parallel computation is nondeterministic in general.
- You can't know which codes finish first (or even if they do!) before running them.
- **Nondeterminism is not necessary** for parallel computation.
  - e.g. purely functional programs can be parallelized automatically.
- **Nondeterminism sometimes gives better performance** in parallel computation.
  - e.g. consumers want to consume values in an order in which they get produced.

Nondeterminism in other languages

`Promise` in JavaScript

```
const foo = async () => {
  // ...
};

const bar = async () => {
  // ...
};

const main = async () => {
  const x = foo();
  const y = await bar();

  (await x) + y;
};
```

Channels in Go

```
func main() {
  c1 := make(chan string)
  c2 := make(chan string)

  go func() {
    c1 <- "fast"
  }()
  go func() {
    time.Sleep(1 * time.Second)
    c2 <- "slow"
  }()

  select {
  case msg := <-c1:
    fmt.Println(msg)
  case msg := <-c2:
    fmt.Println(msg)
  }
}
```

`Promise.race()` in JavaScript

```
Promise.race([compute(x), compute(y)]);
```

Examples in Pen

Futures

- Deterministic parallel computation

```
import Os'Console

f = \(x foo, y foo) bar {
  v = go(\() number {
    computeA(x)
  })

  w = computeB(y)

  aggregate(v(), w)
}
```

Examples in Pen

Racing two futures

- Nondeterministic parallel computation

```
import Os'Console

f = \(x foo, y foo) [number] {
  race([[number] [number computeA(x)], [number computeB(y)]])
}
```

Examples in Pen

Lazy lists (streams or channels)

- Nondeterministic parallel computation

```
import Os'Console

f = \(x foo, y foo) [number] {
  # computeA and computeB produces two series of data computed concurrently.
  race([[number] computeA(x), computeB(y)])
}
```

What else do we need?

- The `go` and `race` built-in functions can represent many concurrency patterns found in other languages.
  - Concurrency in Go
- Circular dependency is apparently impossible to get represented.
  - e.g. actors talking to and depending on each other
- There isn't any research on what primitives for concurrent/parallel computation is necessary for programming languages.
- Existing researches are more about what we can build on the currently available primitives like multi-core CPUs, threads, atomic memory operations, etc.

Summary

- Pen now has two `go` and `race` built-in functions.
- They can represent many concurrent/parallel programming patterns.
- Questions
  - Are they expressive enough in practice?
  - What concurrent/parallel computation primitives are necessary for languages to be expressive enough?
  - Application development?