7章

純粋関数型の並列処理

現代のコンピュータ

- CPUに複数のコアが搭載されている
- 複数のCPUが搭載されている

並列処理を活用する
プログラミングが重要

しかし...

- 並行プログラミングは難しい
- データ共有
- デッドロック

純粋関数型だけで表現すれば
うまくいくのでは？

合成可能性、モジュール性にすぐれた
ライブラリを純粋関数型で作成する

ライブラリ作成は
反復方式で進める

- まずインターフェイスを考える
- インターフェイスと実装の間を行き来しながら
  ユースケースを徐々に複雑にしていくことで
  ドメインと設計空間への理解を深める

7.1

データ型と関数の選択

まずは小さく考える

- 並列化が可能な単純な計算
  - 整数リストの合計

```
def sum(ints: IndexedSeq[Int]): Int = {
  ints.foldLeft(0)(op = _ + _)
}
```

まずは小さく考える

- 分割統合アルゴリズム版

```
def sum(ints: IndexedSeq[Int]): Int = {
  if (ints.size <= 1) {
    ints.headOption.getOrElse(0)
  } else {
    val (l, r) = ints.splitAt(ints.size / 2)
    sum(l) + sum(r)
  }
}
```

並列計算のためのデータ型 (コンテナ型の作成)

- 計算した結果を取り出せる型を作成

```scala
trait Par[A]

object Par {

  def unit[A](a: => A): Par[A] = ???

  def get[A](a: Par[A]): A = ???

  def sum(ints: IndexedSeq[Int]): Int = {
    if (ints.size <= 1) {
      ints.headOption.getOrElse(0)
    } else {
      val (l, r) = ints.splitAt(ints.size / 2)
      val sumL: Par[Int] = Par.unit(sum(l))
      val sumR: Par[Int] = Par.unit(sum(r))
      Par.get(sumL) + Par.get(sumR)
    }
  }

}
```

unit の挙動には選択肢がある

- unit が呼ばれた時点で計算を開始
  - 参照透過が失われる(get が待機してしまう)
- unit には get 限定の副作用がある
  - Par.get(Par.unit(sum(l))) + Par.get(Par.unit(sum(r)))
  - 待っちゃう
- 待たずに結果を結合したい

並列計算の結合

- Par.map2 を定義する

```
object Par {

  def unit[A](a: => A): Par[A] = ???

  def get[A](a: Par[A]): A = ???

  def map2[A, B, C](pa: Par[A], pb: Par[B])(op: (A, B) => C): Par[C] = ???

  def sum(ints: IndexedSeq[Int]): Par[Int] = {
    if (ints.size <= 1) {
      ints.foldLeft()
      Par.unit(ints.headOption.getOrElse(0))
    } else {
      val (l, r) = ints.splitAt(ints.size / 2)
      Par.map2(sum(l), sum(r))(_ + _)
    }
  }

}
```

並列計算の結合

- Par.get を利用せずに結果を結合できるように
- 再起部分から Par.unit が消えている
    - Par.map2 が結合の対象となる2つの計算が独立していて、同時に実行できることを示す
- しかしこのコードにも問題が…
    - Par.map2 の左側から評価(並列計算)がシリアルにされてしまう

明示的なフォーク

- でも Par.map2 の引数を常に同時に評価したいわけではない
  - Par.unit(1) とか即評価されるので不要
- 現在のAPIでは並列計算させるタイミングが曖昧
  - 明示的な Par.fork を作成する
  - 特定のParを別の論理スレッドで利用すべきであることを示す

明示的なフォーク

```
object Par {

  def unit[A](a: => A): Par[A] = ???

  def get[A](a: Par[A]): A = ???

  def map2[A, B, C](pa: Par[A], pb: Par[B])(op: (A, B) => C): Par[C] = ???

  def fork[A](a: => Par[A]): Par[A] = ???

  def sum(ints: IndexedSeq[Int]): Par[Int] = {
    if (ints.size <= 1) {
      ints.foldLeft()
      Par.unit(ints.headOption.getOrElse(0))
    } else {
      val (l, r) = ints.splitAt(ints.size / 2)
      Par.map2(Par.fork(sum(l)), Par.fork(sum(r)))(_ + _)
    }
  }

}
```

ここまでで2つの問題に対応

- 並列タスクの結果を結合すべきであること指定できること
- 特定のタスクを非同期で実行すべきかどうか選択できること
- これらを分けておくことでコンビネータに並列化のグローバルポリシーの類が適用されるの回避できる

新たな問題(fork と get)

- 並列を開始させるのはforkを呼び出した時か、getが呼ばれた時まで遅延させるか
- 別の論理スレッドで実行させるには様々な情報(コンテキスト)が必要
- コンテキストは並列化戦略で利用される
- forkの時点で必要とした場合のようなプリミティブコンビネータで利用されるべきではない

新たな問題(fork と get)

- get にまかせましょう
    - つまり get に並列計算に必要なロジックをもたせるだけでよい
        - Par自体は知らなくてよい
          →Parは実行可能なファーストクラスプログラム
- get を run に変更し、並列化を実装する場所であることを明示する

7.2

表現の選択

現時点のAPI

```
object Par {

  def unit[A](a: => A): Par[A] = ???
  def map2[A, B, C](pa: Par[A], pb: Par[B])(op: (A, B) => C): Par[C] = ???
  def fork[A](a: => Par[A]): Par[A] = ???
  def lazyUnit[A](a: => A): Par[A] = fork(unit(a))
  def run[A](a: Par[A]): A = ???

  def sum(ints: IndexedSeq[Int]): Par[Int] = {
    if (ints.size <= 1) {
      ints.foldLeft()
      Par.unit(ints.headOption.getOrElse(0))
    } else {
      val (l, r) = ints.splitAt(ints.size / 2)
      Par.map2(Par.fork(sum(l)), Par.fork(sum(r)))(_ + _)
    }
  }

}
```

ExecutionServiceをつかったrunの実装

- run が ExecutionService にアクセスできると仮定した場合の Par の表現として、もっとも単純なモデル

```
type Par[A] = ExecutionService => Future[A]
```

- run も簡単になる

```
def run[A](s: ExecutionService)(a: Par[A]): Future[A] = a(s)
```

- (ここでのFutureは、java.util.concurrent.Future)

7.3

APIの改良

現時点のAPIを実装

```scala
object Par {
  type Par[A] = ExecutorService => Future[A]

  def unit[A](a: => A): Par[A] = (es: ExecutorService) => UnitFuture(a)

  private case class UnitFuture[A](get: A) extends Future[A] {
    override def isDone: Boolean = true
    override def get(timeout: Long, unit: TimeUnit): A = get
    override def isCancelled: Boolean = false
    override def cancel(mayInterruptIfRunning: Boolean): Boolean = false
  }

  def map2[A, B, C](a: Par[A], b: Par[B])(f: (A, B) => C): Par[C] = {
    (es: ExecutorService) =>
      val af = a(es)
      val bf = b(es)
      UnitFuture(f(af.get(), bf.get()))
  }

  def fork[A](a: => Par[A]): Par[A] = {
    es => es.submit(new Callable[A] {
      override def call(): A = a(es).get()
    })
  }

  def lazyUnit[A](a: => A): Par[A] = fork(unit(a))

  def run[A](es: ExecutorService)(a: Par[A]): Future[A] = a(es)
```

現時点のAPIを実装

- java.util.concurrent.Future は純粋関数型ではない
  - ライブラリのユーザにFutureを直接使わせるのは避けたい
- java.util.concurrent.Future が副作用に依存しているが、ParのAPI全体は純粋なままである点が重要
  - ユーザは純粋なAPIを扱える
    - 処理の定義と実行する箇所(Par.run)の分離

コンビネータの発展

- 既存のコンビネータで何ができるか

```
def sortPar(parList: Par[List[Int]]): Par[List[Int]] = {
  map2(parList, unit(()))((a, _) => a.sorted)
}
```

- これはさらに一般化できる

- 下記のようなコンビネータを定義

```
def map[A, B](a: Par[A])(f: A => B): Par[B] =
  map2(a, unit(()))((a, _) => f(a))
```

コンビネータの発展

- sortListが単純になる

```
def sortPar(parList: Par[List[Int]]): Par[List[Int]] = map(parList)(_.sorted)
```

- mapはmap2とunitをつかって表現できる
  - mapをつかってmap2はできない
  - map2がmapより強力であるということ
- プリミティブに思える関数が、それより少し強力なプリミティブで表現できるということはよくあること

コンビネータの発展

- 他にもコンビネータを考えてみよう！
  - EXERCISE
    - def sequence[A](ps: List[Par[A]]): Par[List[A]]
    - def parFilter[A](as: List[A])(f: A => Boolean): Par[List[A]
- 他にも
  - map2ベースでmap3、map4、map5など

7.4

APIの代数

APIの代数

- APIに関するここまで議論は便宜的なもの
- それでも問題はないが一歩下がって、
  APIに適用されることを期待する、適用したい
  「法則」を明文化してみる
  - 知らず知らず頭のなかでは期待する法則のイメージを作っている
  - これを実際に書きだして明文化すると便宜的な議論ではみえなかった設計上の選択肢が見えてくる

法則の選択

- 法則の選択には結果が伴う
  - その選択により
    - 演算の意味が制約される
    - 実装上の選択肢が決定される
    - 他のプロパティにも影響が及ぶ

マッピングの法則

```
map(unit(1))(_ + 1) == unit(2)
```

- _ + 1関数によるunit(1)へのマッピングが、unit(2)と同等であることを宣言

- このような有効であると想定される恒等の具体的な例が法則の出発点になる

- 一般化すると

Parにしてからfを適用するのと
fに適用してからParにするので
結果が一致すること

```
map(unit(x))(f) == unit(f(x))
```

マッピングの法則

- これのfを恒等関数(引数をそのまま返す関数)にしてみる

```
map(unit(x))(id) == unit(id(x))
map(unit(x))(id) == unit(x)
map(y)(id) == y
```

- この法則が示しているのはmapのことだけ
- これの違反パターン
  - 関数を結果に適用する前に例外をスロー
  - 計算を途中でやめてしまう

マッピングの法則

- map(y)(id) == y であると仮定すると
  map(unit(x))(f) == unit(f(x)) は真でないといけない
    - この法則定理は、mapのパラメトリシティによって自動的に得られることから「フリー定理」とも呼ばれる
- EXERCISE7.7
    - map(y)(id) == y と仮定した場合、
      map(map(y)(g))(f) == map(y)(f compose g)が成り立つフリー定理である。これを証明せよ。

パラメトリシティとは

パラメトリシティとは

パラメトリシティとHaskellに関する

高階型理論におけるパラメトリシティーの理論
研究課題番号：10780185

代表者 1998年度～1999年度
竹内 泉
研究者番号：20264583
京都大学・大学院・情報学研究科・助手

この研究課題のドキュメント

1998年 研究実績報告書
1999年 採択課題　研究実績報告書

研究課題基本情報(最新年度)

研究期間	1998年度～1999年度		
研究分野	計算機科学	審査区分	
研究種目	奨励研究(A)	研究機関	京都大学
配分額	総額：800千円　1999年度：800千円 (直接経費：800千円)		

研究概要(最新報告)

本研究では、高階型理論におけるパラメトリシティーの理論について研究し、以下の成果を得た。

まず、高階型理論の基礎として、二階型付ラムダ計算の上の二階帰納法とパラメトリシティーの関係を研究した。そして、パラメトリシティーの理論から、各種の帰納法が自然に導入されることを示した。この研究は英文学術雑誌「フンダメンタ・インフォルマチカ」に収録された。

次に、その理論を高階型理論に拡張した。高階型理論では、圏論の概念を幾つか表現することが出来る。圏論に於て随伴函手は重要な道具とされている。「極限を保存する函手には随伴函手がある」という定理は、随伴函手に関して最も基本的な定理である。高階型理論では、函手、極限、随伴函手等の概念を表現することが出来る。その様に表現した許では、パラメトリシティーの理論からこの定理が証明される。このことが、本研究の主要な成果である。

また本研究では、多相型の計算にとって重要である、最汎型付けについても研究した。従来、線型項の最汎型と準線型項の最汎型に対して、その特徴付けがなされていた。その両者の証明は独立に与えられていた。一方、ベラルディは計算項の最適化という観点から「枝刈」という方法を提案していた。本研究では、枝刈の方法によって、両者の証明が本質的には同一であることを明らかにした。この成果は国際学会「2000年オーストララシア計算機科学週間」にて発表し、英文学術速報「理論的計算機科学電子版速報」に収録された。

WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information

Article Talk　Read Edit View

Parametricity

From Wikipedia, the free encyclopedia

In programming language theory, **parametricity** is an abstract uniformity property enjoyed by parame
the intuition that all instances of a polymorphic function act the same way.

Contents [hide]

1 Idea
2 History
3 Parametricity and programming language implementation
4 Parametricity and dependent types
5 See also
6 References
7 External links

Idea [edit]

Consider this example, based on a set X and the type $T(X) = [X \to X]$ of functions from X to itself. The
by $twice_X(f) = f \cdot f$, is intuitively independent of the set X. The family of all such functions $twice_X$, param
polymorphic function". We simply write **twice** for the entire family of these functions and write its type
$twice_X$ are called the *components* or *instances* of the polymorphic function. Notice that all the compon
they are given by the same rule. Other families of functions obtained by picking one arbitrary function

パラメトリシティとは

```
__人人人人人人人__
>　ヤバそう！　<
￣Y^Y^Y^Y^Y^Y^Y￣
```

フォークの法則

- fork が並列計算の結果に影響されないことを考える

```
fork(x) == x
```

- この単純な法則が、forkの実装を強く制約される

- この法則の違反ケースを考えてみる

法則への違反：バグ

- fork(x) == x の x
  - fork、unit、map2、およびそこから派生するコンビネータを利用する式
- 実はいまの fork の実装には問題がある

```
def fork[A](a: => Par[A]): Par[A] = {
  es => es.submit(new Callable[A] {
    override def call(): A = a(es).get()
  })
}
```

法則への違反：バグ

- 有限サイズのスレッドプールに関連付けられている ExecutorService を使用すると簡単にデッドロックしてしまう

```
val a = lazyUnit(42 + 1)
val S = Executors.newFixedThreadPool(1)
println(Par.equals(S)(a, fork(a))
```

法則への違反：バグ

```
def fork[A](a: => Par[A]): Par[A] = {
  es => es.submit(new Callable[A] {
    override def call(): A = a(es).get()
  })
}
```

- Callable の call が別スレッドから呼ばれ、
  a(es).get の部分で待ちが発生するが、
  a(es) 内部も別スレッドが必要なため、
  スレッドプールのスレッド数が1だと永遠に処理
  されずデッドロックになる

法則への違反：バグ

- 固定サイズのスレッドプールだと発生してしまう
- 回避方法は2つ
  - 際限なく拡張することが可能なスレッドプールが必要であることを明文化する
  - 隠れていた不文律や前提を明文化せざるを得なくなるため十分な効果がある
- 固定サイズのスレッドプールに対応するようにforkを修正する

```
def fork[A](a: => Par[A]): Par[A] =
  es => a(es)
```

法則への違反：バグ

- fork の実装を修正する方法について
  - たしかにデッドロックは回避できる
    - しかし別の論理スレッドをフォークできていない
- 我々が望んでいるのは固定サイズのスレッドプールで任意の計算を実行できるようにすること
  - Par の別の表現を選択する必要がありそう

アクターを使ったParの完全なノンブロッキング実装

- Future.get を呼び出すとスレッドをブロックしてしまう
- 先ほどの fork や map2 でカレントスレッドをブロックしてしまうのはダメなので、ノンブロッキングである必要がある
- そうだ、ノンブロッキングなFutureを自分で実装しよう
  - 結果が準備できたときに呼び出されるコールバック(継続:continuationともいう)を登録できるようにしよう

アクターを使ったParの完全なノンブロッキング実装

```
trait Future[A] {
  private[ss7] def apply(k: A => Unit): Unit
}

type Par[+A] = ExecutorService => Future[A]
```

- apply(k: A => Unit): Unit の k が継続部分
  - Future の結果である A を期待している
- apply メソッドはパッケージprivateになっているのでユーザに提供されず、APIの純粋性が保たれる

アクターを使ったParの完全なノンブロッキング実装

- この Future で run 関数を書きなおしてみる

```
def run[A](es: ExecutorService)(p: Par[A]): A = {
  val ref = new AtomicReference[A]()
  val latch = new CountDownLatch(1)
  p(es) { a => ref.set(a); latch.countDown() }
  latch.await()
  ref.get()
}
```

- latch を待機している間は呼び出し元スレッドはブロックされる

  - ブロックしない run は実装できない

アクターを使ったParの完全なノンブロッキング実装

- この Future で色々書きなおしてみる
  - unit

```scala
def unit[A](a: A): Par[A] =
  es => new Future[A] {
    def apply(k: A => Unit): Unit =
      k(a)
  }
```

  - fork

```scala
def fork[A](a: => Par[A]): Par[A] =
  es => new Future[A] {
    def apply(k: A => Unit): Unit =
      eval(es)(a(es)(k))
  }

def eval(es: ExecutorService)(r: => Unit): Unit =
  es.submit(new Callable[Unit] { def call = r })
```

- map2 は結構難しいらしい

アクターを使ったParの完全なノンブロッキング実装

- map2 をアクターを使って実装してみる
- アクター(Actor)とは
  - ノンブロッキングの並列化プリミティブ
  - メッセージを受け取った時だけスレッドを専有する
  - 複数のスレッドから同時にアクターにメッセージを投げれるが、アクターが処理するメッセージは1つずつ
    - 競合状態やデッドロックを回避できる

アクターを使ったParの完全なノンブロッキング実装

```
def map2[A, B, C](pa: Par[A], pb: Par[B])(f: (A, B) => C): Par[C] =
  es => new Future[C] {
    def apply(k: C => Unit): Unit = {
      var ar: Option[A] = None
      var br: Option[B] = None
      val combiner = Actor[Either[A, B]](es) {
        case Left(a) => br match {
          case None => ar = Some(a)
          case Some(b) => eval(es)(k(f(a, b)))
        }
        case Right(b) => ar match {
          case None => br = Some(b)
          case Some(a) => eval(es)(k(f(a, b)))
        }
      }

      pa(es)(a => combiner ! Left(a))
      pb(es)(b => combiner ! Right(b))
    }
  }
```

法則の選択まとめ

- この節は法則の重要性を示した
- ライブラリが提供する機能のプリミティブな関数に対して、有益と思われる法則や啓蒙的な法則を洗い出し、モデルに適用することが可能
- 実装を調べて、その結果に基いて有効であると期待される法則を推測することもできる

7.5
コンビネータを最も汎用的な形式に改良する

コンビネータを最も汎用的な形式に改良する

- 関数型の設計は反復的な作業
  - プロトタイプ実装後は、シナリオの複雑さや現実性を徐々に引き上げながら使ってみる
  - シナリオによっては新しいコンビネータが必要になるかも
    - 作業に取り掛かる前に、必要なコンビネータを最も汎用的な形式に改良できるかどうかを調べてみる

コンビネータを最も汎用的な形式に改良する

- 例 `def choice[A](cond: Par[Boolean])(t: Par[A], f: Par[A]): Par[A]`
  - 最初の計算結果(cond)に基いて、フォークする2つの計算のどちらかを選択する関数
  - 一見良さそうに見えるがしかし、不自然な点も
    - 並列計算の選択肢がなぜ2つ？
    - N個の計算から選択できるのも便利なはず

`def choiceN[A](n: Par[Int])(choices: List[Par[A]]): Par[A]`

コンビネータを最も汎用的な形式に改良する

- N個の計算から選択できるやつを作ってみる

```
def choiceN[A](n: Par[Int])(choices: List[Par[A]]): Par[A]
```

- choice は choiceN で実装できる！
- より汎用的になった
- さらに汎用的にしてみよう
  - Exercise7.13(Exercise7.12は噛ませ犬)
    - 次を実装する

```
def chooser[A, B](pa: Par[A])(choices: A => Par[B]): Par[B]
```

コンビネータを最も汎用的な形式に改良する

- ここまで一般化した時点で、一度批判的に関数を見てみる

- その関数をつくるきっかけとなったのは特定のユースケースだが、シグネチャと実装には、より汎用的な意味があるかも

コンビネータを最も汎用的な形式に改良する

- chooser はもっと最適な名前がある
  - 1つ目の計算が実行され、その結果から2つ目の計算が決定される、または作り上げられる
- このような処理は関数型ライブラリでは「bind」または「flatMap」という名前が付いている

```
def flatMap[A, B](a: Par[A])(f: A => Par[B]): Par[B]
```

コンビネータを最も汎用的な形式に改良する

- flatMap は本当に最もプリミティブな関数か？
  - flatMap は2ステップに分解できる
    - Par[A] に f: A => Par[B] を適用させて、Par[Par[B]] を生成する
    - Par[Par[B]] を Par[B] にフラッティング(平坦化)する
- map はあるので、次のコンビネータを追加するだけでよい

```
def join[A](a: Par[Par[A]]): Par[A]
```

7.6

まとめ

まとめ

- 関数型の設計プロセスは反復型
- その代数のプリミティブな演算(関数)を発見し、それを用いてコンビネータを作成していく
- プリミティブな演算の発見には、その代数が関わるであろう法則を見つけ、明文化するのがよい
- この章では、読者がこのような設計をしていく中で陥りがちな問題を明らかにし、どのように対処していけばよいかを示した