関数プログラミング班成果報告書

小林賢治[1]　　佐々木章徳[2]　　廣田大地[3]

2017年08月09日

[1]情報理工学部情報システム学科2回生
[2]情報理工学部情報システム学科3回生
[3]情報理工学部1回生

目 次

第1章　はじめに

佐々木 章徳

このプロジェクトは関数プログラミングを研究するプロジェクトである．関数プログラミングとはラムダ計算の理論をもとにしたプログラミングパラダイムであり，四則演算や代入を主な操作とする手続き型とは違い関数の適用をプログラムの主な操作とする．

関数プログラミングを用いたコードは参照透過性により関心の分離がしやすく，またその数学的性質によりプログラムの論証がしやすいなどの特徴を持つ．現在では様々な言語にこのパラダイムが取り入れられており，これからの時代に必須の知識となることが予想される．

1.1　活動形態

このプロジェクトは以下の二つの活動によって構築される．

- JavaScript による関数プログラミング
- Haskell におけるアクターモデル DSL の構築

この報告書はそれぞれの活動の 2 つの章によって構成される．

第2章　JavaScriptによる関数型プログラミングと処理の速度向上

小林賢治 廣田大地

2.1　目標

本プロジェクトでは，”関数型プログラミングを学び利点等を正しく理解し， 実装をできること ”を目標とした．なお， 本プロジェクトは JavaScript というプログラミング言語を使用した．JavaScript は関数型プログラミングとオブジェクト指向プログラミングの両方を実装可能であり，近年オブジェクト指向プログラミングから関数型プログラミングに移り代わりつつある言語である．そのため，両アルゴリズムの利点と問題点を理解するのに最適な言語であるという考えから本プロジェクトでの採用に至った．

2.2　本プロジェクトのゴール

本プロジェクトの目標である ”関数型プログラミングを学び利点等を正しく理解し，実装をできること ”を満たすために，本プロジェクトのゴールは関数型プログラミング向けのライブラリを実装することとする．しかし，既にライブラリが存在するため本プロジェクトでは既存ライブラリよりも高速で処理することもゴールに追加する．

2.3　動作環境について

本プロジェクトで実装するライブラリは V8 エンジン向けに開発する．V8 エンジンとは，Chrome， Chromium， Opera， Node.js で使用されている C++を中間言語とし，JavaScript から機械語へのコンパイルを行う C++製のコンパイラである．V8 エンジンには C++の実行環境があり，C++製のライブラリ等を V8 エンジン経由で Javascript から読み込み実行が可能である．この機能を使用し，C++で JavaScript 向けモジュールを実装する．

2.4　関数型プログラミングの基礎学習

まず最初に，関数型プログラミングの学習を行った．入門サイトや，サンプルプログラミングでの学習を主とした．

2.5 実装関数の選択

V8 エンジンで実装するために，関数型プログラミングにおいて用いられる関数の動作確認を行った．今回は時間の関係により，以下の関数のみ詳細な確認を行った．

- map
- each

よって，上記の関数のみを実装する．

2.6 結果

選択した関数は実装することができた．また，処理速度については，JavaScript のモジュールと今回作成した C++のモジュールの速度比較を行った．結果としては，読み込み速度，実行速度共に明確な速度差を観測することができなかった．しかし，関数とは異なる文字列に対する処理を実装し実行したところ，C++のほうが平均して 0.5ms ほど早かったので，大量の関数やより複雑な処理を行うことで明確な速度差を観測できるのではないだろうか．

2.7 まとめ

時間的な問題もあり，今回は少数の関数の実装までしか行うことができなかった．残った課題として，以下の点が挙げられる．

- 大量の関数を実装したライブラリの作成
- 上記ライブラリを用いたものと既存フレームワークの関数レベルでの速度比較
- 上記ライブラリを用いたものと既存フレームワークのソフトウェアレベルでの速度比較

また，関数の実装に重きを置きすぎて関数型自体の理解に不安が残った．しかし，関数の動作で学習することで，非関数型言語において関数型プログラミングを可能にする事ができるという知見を得ることができた．よって，関数型プログラミング班としての進捗は十分にあったと言えるだろう．

第3章　Freeモナドによるアクターモデル DSL の構築

佐々木 章徳

3.1　はじめに

Haskell という純粋関数型プログラミング言語において，モナドという概念は非常に一般的である．Haskell ではモナドを用いることで手続きを純粋関数プログラミング上で実現する．そのモナドの中でも，ある型にモナドの構造を持たせる機能を持つ Free モナドというものが存在する．この Free モナドを使うことによって，Haskell のプログラム内で利用できる手続き言語を構築することが可能となる．

今回はその Free モナドを用いてアクターモデル並列計算用 DSL を実装した．このアクターモデル DSL を利用することによって，並行・並列プログラミングを容易に記述し，マルチコア CPU の性能を生かすことを目的としている．

3.2　モナド

モナドとは，圏論という数学に由来する概念であるが，ここでは深く考えず関数型プログラミングにおいて手続きを導入する仕組みとして扱う．

純粋関数型である Haskell では，プログラムが関数の適用によってのみ構成されているので，基本的に手続きを記述することができない．

しかし，プログラムにおける処理が本質的に手続きを必要とする場合や手続きを用いたほうが簡潔に書ける場合，その処理は手続きを用いて書くほうがよい．

Haskell では，モナドを用いることで特殊な手続きを記述できる．例えば，状態を扱うモナドとして State モナドがある．これを用いると，ある変数の状態を扱う手続きを以下のように書ける．

```
-- スタックの構造を表す型
-- 中身は片方向リスト
type Stack a = [a]

-- スタックの操作手続き
pop :: State (Stack a) a
pop = do
  (x:xs) ← get
```

```
  put xs
  return x

push :: a → State (Stack a) ()
push x = modify (\xs → x:xs)

-- スタック上の要素を用いた演算
-- はスタックから要素をつポップしてadd2
-- それらを足したものをプッシュする
add :: State (Stack Int) ()
add = do
  x ← pop
  y ← pop
  push (x+y)

-- ではポップしたつの要素を引き算したものをプッシュsub2
sub :: State (Stack Int) ()
sub = do
  x ← pop
  y ← pop
  push (y-x)

-- 単純な逆ポーランド記法の数式を解釈するプログラム
prog :: String → State (Stack Int) Int
-- すべての式を読んだらスタックの一番上のデータを取り出す
prog "" = pop
prog (x:xs) = do
  if (isDigit x) then
    -- 数値の時はそれをスタックに追加
    push (toInt x)
  else if (x=='+') then
    -- '+' の時はスタックの上の二つを加算
    add
  else if (x=='-') then
    -- の時はスタックの上の二つを'-'
    sub
  else
    -- それ以外の時は何もしない
    return ()

  -- 残りの式を解釈
```

```
prog xs
```

上のプログラム prog は，一つの状態 (Int 型のデータを扱うスタック) のみを扱うことができ，それ以外の状態は操作できないという特徴がある．

一つの状態のみしか扱えないという性質により，以下のような利点がある．

1. 誤っての状態を変更することがないため安全である
2. 一つの状態を扱うことのみに集中できる

このように，モナドによって構築された手続きは，ある目的に特化したものとなっている．ある一つの目的に特化した言語をドメイン特化言語 (DSL) というが，モナドによって構築される手続きは言語内で使える DSL という側面を持っている．

3.3 Free モナド

Free モナドとは，以下の構造によって定義されるデータ構造である．

```
data Free f r = Pure r | Free (f (Free f r))
```

この Free 型は型パラメータ f にファンクタ (関数を適用可能というモナドよりも弱い性質) の型をとることでモナドとなる．

```
instance Functor f => Monad (Free f) where
  return = Pure
  (Pure a) >>= f = f a
  Free m >>= f = Free (fmap (>>= f) m)
```

つまり，任意のファンクタの型をモナドに持ち上げることができる．

これを用いることによって，コマンドを表すファンクタの型を用意するだけで容易に言語内 DSL を構築することができる．

3.4 アクターモデル

アクターモデルとは並行計算モデルの一つである．アクターモデルを用いたプログラムはアクターと呼ばれるそれぞれ並行に動作する計算実体によって構成される．

それぞれのアクターはメッセージキューを持っており，アクター同士がメッセージを送りあうことで処理を実現する．アクター間では共有データはなく，メッセージパッシングによってのみデータを送信する．そのため共有メモリのアクセス時に必要なロックが不必要となる．

この計算モデルを採用しているプログラミング言語としては Erlang や Pony などがある．

3.5 実装

実装したアクターモデル DSL のソースコードは `https://github.com/r-edamame/haskell-actor` にアップロードしているので必要があればそれを参照されたい．

3.5.1 アクションの定義

この DSL におけるアクションを定義した Functor の型 Actor' rec a は以下のようになっている.

```
data Actor' rec a
  = Receive (rec → a)
  | Self (MailBox rec → a)
  | EmbedIO (IO a)
```

Receive

Receive では，そのアクターの持つメッセージキューから受信したメッセージを一つ取り出すという操作を提供する.

Self

Self アクションでは，そのアクターが持つメッセージキューへの参照を取得するという操作を提供する.

EmbedIO

上記二つのアクションだけでは入出力を行うことができないため，EmbedIO というアクターモデル DSL の中に IO 処理を書くことのできるアクションを用意した.

3.5.2 モナド化

Functor である Actor' に対して Free を用いてモナドの構造を与えることで以下の型 Actor rec a はモナドとなる.

```
type Actor rec a = Free (Actor' rec) a
```

これが実際にアクターモデル DSL として使用する型となる.

ここで，Actor がとる型引数として rec と a がある．rec はそのアクターが受け取るメッセージの型を表しており，a はそのアクションの結果の型を表している．次に，Actor' のそれぞれのアクションを Actor の型に変換し，DSL における手続きとして使えるようにする.

```
receive :: Actor rec rec
receive = Free (Receive Pure)

self :: Actor rec (MailBox rec)
self = Free (Self Pure)
```

```
embedIO :: IO a → Actor rec a
embedIO m = Free (EmbedIO (return <$> m))
```

また，アクターモデルにおける基本操作としてのメッセージの送信を embedIO を用いて実装しておく．

```
send :: MailBox b → b → Actor rec ()
send mb m = embedIO (sendMail mb m)
```

解釈関数

Haskell におけるプログラムの実行主体は IO であり，Actor はそれ自体で実行することはできない．よって，この Actor を IO に変換し実行するための解釈関数を定義する．

```
runActor :: MailBox rec → Actor rec a → IO a
runActor mb (Pure a) = return a
runActor mb (Free (Receive f)) = readMail mb >>= runActor mb . f
runActor mb (Free (Self f)) = runActor mb (f mb)
runActor mb (Free (EmbedIO m)) = m >>= runActor mb
```

これによって Actor から IO への変換ができ実行できるようになった．

また，これを用いてアクターモデルでの操作である子アクターの生成 (spawn) と新しいメッセージボックスの生成を隠蔽した Actor から IO への変換 (start) を定義する．

```
spawn :: Actor rec' () → Actor rec (MailBox rec')
spawn actor = do
  mb ← embedIO newMailBox
  embedIO (forkIO (runActor mb actor))
  return mb

start :: Actor rec a → IO a
start actor = do
  mb ← newMailBox
  runActor mb actor
```

spawn の中で使われている forkIO は新たなスレッドを生成する IO アクションで，これにより並行したアクターの実行を達成できる．

3.6 利用方法

このアクターモデル DSL の利用方法を並列計算を模した以下のサンプルコードで示す．

```
data Work = Work Int (MailBox Int)

worker :: Actor Work ()
```

```
worker = do
  (Work param sender) ← receive
  embedIO $ threadDelay (param * 10^6)
  send sender param

main :: IO ()
main = start $ do
  -- ワーカーアクターの生成
  worker1 ← spawn worker
  worker2 ← spawn worker
  worker3 ← spawn worker

  s ← self

  -- 仕事の割り当て
  send worker1 (Work 2 s)
  embedIO $ putStrLn "send work 2"
  send worker2 (Work 3 s)
  embedIO $ putStrLn "send work 3"
  send worker3 (Work 4 s)
  embedIO $ putStrLn "send work 4"

  -- 結果の取得
  replicateM_ 3 $ do
      res ← receive
      embedIO $ putStrLn $ "received result : " ++ show res
```

3.6.1 worker

ここではworkerというアクターを定義している．このworkerはWorkというInt型のデータと返信先をまとめた型のデータを受け取る．処理の中身として与えられた整数分スレッドを遅延させ，時間のかかる計算を表現している．その後返信先のメールボックスに結果を送信する．

3.6.2 main

Haskellでのmainの型はIO ()なので，start関数を用いることでmainの中にアクターDSLを記述することを可能にしている．

mainアクターの中でworkerアクターを3つ生成し，それぞれに2秒，3秒，4秒の時間がかかる処理を与えている．その後結果を受け取り，それぞれ表示する．

3.6.3 実行時間

このプログラムの実行時間は約 4 秒であった．この処理を直列に実行すると約 9 秒かかることになるので，並列化することによって高速化が行えているといえる．

今回は実際の計算を含まずスレッドの遅延によって処理を表しており実際どの程度高速化されるかは不明であるが，CPU のコア数が十分にありすべてのスレッドに CPU が割り当てられれば十分な高速化が見込めると考えられる．

3.7 まとめ

Free モナドを利用することによって簡単に言語内 DSL を作成することができ，アクターモデルを用いたプログラムを手続きとして記述可能になった．しかし，性能面については評価を行えていないので実用性は不明である．また，機能面に関しても子アクターの管理や例外への対応など足りないものも多い．これらの問題を解決しより使いやすくしていくことを今後の課題として取り組んでいきたい．