プログラム言語特論 前半レポート

コンピュータサイエンス専攻 江畑 拓哉 (201920631)

1 Functional Pearl: A SQL to C Compiler in 500 Lines of Code

本論文における段階的プログラミングの使い方について、自分が理解したところを説明しなさい。

本論文では、特に二村射影に基づいてクエリを静的データとして扱い、データファイルを動的な入力とみなすことで SQL のコードを高速に処理します。その際に Scala のインタプリタを LMS(Lightweight Modular Staging) を用いてステージングすることでコンパイラのような機能を持たせています。

LMS はマルチステージングを行うための Scala ライブラリです。ステージングとはプログラムをプログラムを生成するプログラムにすることを指しています。この際に型を持つことでエラーを防ぐことが出来ます。例えば Int を静的な Int 値であるのに対して、Rep[Int] という動的な Int 値を設けることで、エラーを防ぐことが出来るステージングを提供することが出来ます。ここで SQL の話に戻すと、先述したように、クエリを静的な型を持つ値、データファイルを動的な型を持つ値として扱い、インタプリタをクエリコンパイラにすることが出来、結果として実行速度を向上することが出来た、ということです。

例えば以下のコードを見たとき、fields への型は Vector[Rep[String]] ですので、動的な String 型の Vector となり、この値が操作される計算は動的なものとなります。対して schema の型は Vector[String] ですので、静的な String 型の Vector となり、コード生成時には計算が済まされています (これにより実行時のパフォーマンスを向上させることが出来ます)。これは部分評価の一種としてみなすことが出来ます。この部分評価として本論文が特に注目したものが二村射影になります。二村射影については論文の概要を説明した後に記述します。

```
1  case class Record(
2    fields: Vector[Rep[String]], schema: Vector[String]) {
3  def apply(name: String): Rep[String] =
4      fields(schema indexOf name)
5  }
```

以上のステージング技術によって、手書きした特定の高速な Scala コードを SQL 文から生成することに成功しました。またこの手法によって、特定の高速なコードをそれぞれ作る必要はなく、同じ LMS のコードから生成することが出来るため汎用性を得ることも出来ます。

更にこれの高速化としては、本論文が取り上げているのは、クエリのハッシュ化と列指向データベースの構成、そしてデータのメモリマッピングとそれに伴うデータ要素のクラスの変更です。これらは内部的な変更で、元の SQL の実行と等しい命令文からより高速に処理することが出来ました。

以上のような実装と評価の結果、論文の著者らは次のようなことを主張しています。

一つに、抽象化を行う際に特殊化されたコードに比べたことで後悔しないようにしよう、換言すると、高水準言語で高いパフォーマンスを達成できるようにしよう、ということです。これはもっと突き詰めれば、C 言語のような低水準の言語でなければ高速な処理は望めないという意識を断ち切るという意味にも聞こえます。高水準な言語としては例えば Racket や Haskell、Python、Lua、そして LMS のようなステージングをサポートするそれらの言語のライブラリが挙げられています。

もう一つに関数合成のための高階関数や、ステージングを行うデータ構造を支援するクラス、プログラム生成のモジュール性、これらの有用性についてです。これらはあるドメインでは有効であることが、例えば本論文などで示されていると主張しています。しかし実用的なプログラム生成の技術はまだ発展途上にあるため、言語機能やプログラミングモデルに対してこれらの技術をどのように活かすことが出来るのか、ということを PL コミュニティで活発に議論されることを提案しています。

論文の概要を説明しましたので、二村射影について説明します。二村射影とは第一二村射影と第二二村射影、第三二村射影の三つからなります。

第一二村射影とは入力データとソースコードをインタプリタに通し出力を得るというプロセスに対して、ソースコードをインタプリタと共に第一二村射影によって特殊化しておき、入力データを特殊化したそれに通すことでインタプリタを通した出力と同じ結果を得られる射影を指しています。この特殊化されたそれは C 言語で言うならば実行可能ファイルのようなものとして捉えることが出来ます。これはソースコードから実行可能なもの (C 言語で言うならば実行可能ファイル) への射影として捉えることが出来ます。

第二二村射影とはインタプリタの特殊化を行いコンパイラを得る射影を指します。これはインタプリタとコンパイラを繋げる射影と捉えることが出来ます。

第三二村射影とは第二射影の特殊化を行うことで、インタプリタから自動的にコンパイラを生成するプログラムを生成できる射影、つまりは第二射影からコンパイラ生成プログラムへの射影を指します。

尚、本論文では第一二村射影のみが用いられています。つまり LMS のソースコードの静的な部分についてステージングを行うことで、その実行可能なものを得る処理を行っています。これによってクエリ処理が、動的な値をその実行可能なものに通す処理へと置き換わり実行速度の向上に繋がっています。

この論文についてまず言及しなければならないことは、これは MySQL のようなデータベースとの互換性を持たない、という点です。このロジックでは Insert や Delete、Update に関しての機能を十分に追加できるとは思えず、またトランザクション処理などについても深く言及されていません (論文中で文字としては存在してますが、この機能について実装をしなければならない、という意識を感じることが出来ません)。そのため論文のタイトルは「SQL の検索に対してのプログラム生成技術を用いた高性能な汎化」辺りにする

べきなのかと思いました。というのも、例えばRow-oriented databaseを採用した理由として、メモリ効率が良い、という風に論文では主張し、これを実装していますが、これは列に関する検索に対してメモリ効率が良いのであって、少数のColumnに関するUpdate処理とトレードオフな関係にあるため、前者だけを切り取って主張することは問題があるように思います。

とはいえこの考え方そのものは非常に有用で、例えば(特に検索に関する)スケールアップが容易なImmutableなデータベースとして一時期Lisp界隈有名になったDatomicというクラウドデータベースのような場合では、書き込みを行うTransactorと検索を行うPeerが完全に分離されているため、このPeerに対してこの論文の技術は非常に価値があると考えられます。

次にステージングについて言及すると、この論文(LMS)のステージングにおいて使われる静的な値と動的な値を区別して持つことでコードがわかりやすくなっていることを理解しました。元々型についてあまり知識や理解の足りていないため型を持つ必要性から探る必要がありましたが、今回に関しては型がないと実装が困難であることがわかりました。とはいえLMSやScalaのコンパイラがインタプリタでどこまで実行するのか(バイナリになるのかJVMの中間言語になるのか)がわからないため、その部分についてもより深く研究すればもっと速く処理することが出来るような気がします。(SQLとの互換性の問題から、実験概要・目的について深く納得することが出来なかったこと、Scalaのどのバージョン(言語仕様)を軸にして実験をしているのかよくわからなかったことのため断言は出来ません。)

また二村射影について言及すると、第一二村射影について考えたJITコンパイラとして挙げられるものとして、Oracle社の出したGraal(GraalVM)について考えることが出来ます。これはPythonやJava、JavaScriptなどの中間言語を作成するJITコンパイラで、これを上手くステージングと組み合わせれば、後述するプログラム生成についての議論、というものをより活発化させることが出来ると思います。恐らく本論文ではGraalが登場していなかったので言及されていないですが、LMSとGraalとの組み合わせによる更なるパフォーマンスの向上については研究してみる価値があると思います。これが出来るとGraalVMが対応しているPythonやJavaScriptと言った広い市場を掴んでいる言語でも同様の試みが取り組まれる可能性が見込めると考えています。

最後にプログラム生成についての議論の活発化について言及します。これは現在のところ主要な言語ではマクロやプログラム生成よりもAbstract classやinterfaceの方がデザインパターンやオブジェクト指向に適しているという流れから来ていると思われます。ところが最近のデータ解析や機械学習、Webなどの技術には積極的に関数型の考え方を取り組もうとしていることを考えるならば、プログラム生成について考える機会は十分にあると思います。オブジェクト指向の型と(OCamlやHaskellという意味での)関数型言語の型を同じく考えることが私には出来ていませんが、これらを繋げるか、スムーズな移行をサポートできるならばこれらの研究は十分に活発なものとなると思います。繋げる方法としてはCommon LispのCLOSのように完全にオブジェクト指向を仕様に組み込んだ関数型言語を作る(実際Common LispにはCLOSを活用した機械学習ライブラリMGLが存在しています。(開発はほぼ停止しています))か、ScalaやClojureのようにJava(ここではオブジェクト指向の一例としてみなしている)の機能を用いることが出来る言語を作成するなどがあり、これを皮切りに研究する魅力を増やすことが出来るのではないかと思います。

2 「プログラムによってプログラムを生成する」考えが有効に活用できる場面を説明せよ

Clojure の threading macro はプログラムの可読性を向上させ、編集を容易にし、美しいコードを書くために非常に役に立ちました。

2.1 Threading macro

例えば以下の要件を満たすような関数 transform-list を作成したいと定義します。

1. 入力は可変長リスト
2. 入力要素はまず絶対値を取られる
3. 次に値は対数変換される
4. 次に値は 2 乗される
5. 次に値は 3 倍される
6. 次に値を value の値とする。つまり小数から ArrayMap オブジェクトにする
7. 最後にそれらをリストにして返す

まず以上の機能を記述通りに Clojure で書くと以下のようになります。

```
;; *clojure-version*
;;  => {:major 1, :minor 10, :incremental 0, :qualifier nil}
(defn transform-list-v1 [l]
  (let [square (fn [x] (* x x))
        do-element (fn [x]
                     (array-map :value
                        (* 3
                         (square
                           (Math/log
                             (Math/abs x))))))]
    (map do-element l)))
(transform-list-v1 [1 2 3 4 5 6 7 8 9 10])
```

```
({:value 0.0}
 {:value 1.441359041754604}
 {:value 3.620846882437746}
 {:value 5.765436167018416}
 {:value 7.770871181940704}
 {:value 9.631205986705204}
 {:value 11.359698924589415}
 {:value 12.972231375791434}
 {:value 14.483387529750985}
 {:value 15.905694331435198})
```

do-element 関数に注目して頂きたい。この関数は値を順に手続きの関数に適用しているわけですが、Lisp のシンタックス上、データの流れを追おうとすると、どうしても下から上へ視線を動かさなければなりません。またもし 4 の工程を削除したいときに、編集がやや煩雑になります。このため同じ機能を持つがより可読性の高いマクロを Clojure は提供します。以下がそれを用いた例です。

```
;; *clojure-version*
;; => {:major 1, :minor 10, :incremental 0, :qualifier nil}
(defn transform-list-v2 [l]
  (let [square (fn [x] (* x x))
        do-element (fn [x]
                     (->> x
                          Math/abs
                          Math/log
                          square
                          (* 3)
                          (array-map :value)))]
    (map do-element l)))
(transform-list-v2 [1 2 3 4 5 6 7 8 9 10])
```

```
({:value 0.0}
 {:value 1.441359041754604}
 {:value 3.620846882437746}
 {:value 5.765436167018416}
 {:value 7.770871181940704}
 {:value 9.631205986705204}
 {:value 11.359698924589415}
 {:value 12.972231375791434}
 {:value 14.483387529750985}
 {:value 15.905694331435198})
```

これは Threading macro というマクロで、今回のような関数が連続的に適用される場合にデータの流れを見やすくするために作られています。これは完全に元のコードと同義であり、実行時にはこれは上の形に展開されて実行されます。これによってデータフローを負いやすくなる他、コード量を（括弧の量とはいえ）減らすことができ、編集を容易にしています (必要のない関数の名前を消すだけで良くなります)。

Clojure は Web サーバなどを開発する際に需要の高い言語である。そのためデータ駆動な処理が比較的に多くなる傾向にあります。こうしたときにデータの流れを追いやすいマクロを提供することは、コードやアルゴリズムのミスを減らすことができると考えられます。

補足として、threading macro の展開について示します。

```
(macroexpand
 '(->> x
```

```
        Math/abs
        Math/log
        square
        (* 3)
        (array-map :value)))
;; => (array-map :value (* 3 (square (Math/log (Math/abs x)))))
```

2.2 Lisp のマクロについて

Lisp はマクロとは強く結びついてきた言語で、表現力を拡張したり、DSL を設計したりとプログラム生成とも関連性がある言語とも言えます。しかししばしば Lisp の初心者は、マクロと関数の違いについて疑問に思うことがあります。このため補足としてマクロと関数について紹介します。

Lisp のマクロを理解する最もわかりやすい例として、if-not 関数を挙げることが出来るため、これを例に用います。まず関数版の if-not を my-if-not-func、マクロ版の if-not を my-if-not-macro とし実装します。

```
;; if-not function
(defn my-if-not-func [form then else]
  (if (not form) then else))

;; if-not macro
(defmacro my-if-not-macro [form then else]
  (list 'if (list 'not form) then else))
```

次にマクロがインタプリタ (REPL) に読み込まれることによってどのように展開されるかを示します。

```
(macroexpand
'(my-if-not-macro (zero? x) (println "not zero") (println "zero")))
;; => (if (not (zero? x)) (println "zero") (println "not zero"))
```

これは my-if-not-function と完全に等価であることがわかります。しかしこれを実行すると 2 つは全く違う挙動をします。

```
(my-if-not-func (zero? 1) (println "not zero") (println "zero"))
;; => not zero
;; => zero
(my-if-not-macro (zero? 1) (println "not zero") (println "zero"))
;; => not zero
```

これが起きる原因は、遅延評価という概念を考える必要があります。関数は引数を受け取ったとき、その引数の値を先に評価して関数内部でその値を評価します。しかしマクロは引数をそのままマクロ内部の指定位置に置き換えて、実行時に展開したマクロを実行します。そのため例えばプリント関数などを用いた場合、関数では引数の関数部分が先に評価されてしまい出力が期待したものとは異なってしまい、マクロでは後で評価され期待したものを得られます。これを応用することで Lisp は柔軟なコード生成を行うことが出来、例えば HTML や SQL のコードを Lisp のコードから生成することが出来ます。

ちなみにこのマクロの問題点 (利点でもある) は以下のようなコードが通ってしまう点です。

```
;;(my-if-not-func (zero? 1) (println "not zero") (println (/ 1 0)))
;; => not zero
;; => class java.lang.ArithmeticExceptionclass
;;      java.lang.ArithmeticExceptionArithmeticException
;;          Divide by zero  clojure.lang.Numbers.divide
;;              (Numbers.java:158)
(my-if-not-macro (zero? 1) (println "not zero") (println (/ 1 0)))
;; => not zero
```