実行時コンパイルを用いた正規表現評価器の実装

新屋 良磨

当研究室では, Concinuation based C という, 状態遷移記述に適した C の下位言語を提案している.Continuous bsed C は ステートメントより大きく, 関数よりも小さなプログラミング単位としてコードセグメントを持ち, コードセグメントからの継続を基本としている. 本研究では, 与えられた正規表現から, 等価な 有限状態オートマトンに変換し, オートマトンにおける状態遷遷移を Continuous based C による継続に変換する正規表現コンパイラ を Python で実装した. なお, ここで言うコンパイルは, 内部形式/中 間表現への変換だけでなく, 実行時バイナリの生成までを指す.

1 はじめに

近年, 実行時コンパイルによる高速化 (Just-in-Time Compile) が様々なプログラムで用いられている. これらは, コンパイラ理論の発展により実行時コンパイルにかかるオーバーヘッドよりも, コンパイルによって得られる機械語レベルのプログラムの実行速度が上回る場合において有効であり, たとえば Java の HotSpot や Python の PyPy など, 仮想マシンを持つ言語処理系の最適化技術として利用されている.

実行時コンパイルが可能な対象として, 正規表現評価器に着目した. 現在, 正規表現の評価器は, プログラミング言語の組み込み機能やライブラリ等, さまざまな実装が存在するが, それらの殆どは仮想マシン方式を採用している [2]. 仮想マシンを採用いた実装でも, 正規表現を内部表現に変換する処理を行っており, それらを “コンパイル” と呼ぶことが多い. 本研究で実装した評価器の “実行時コンパイル” とは, 正規表現を内部形式に変換することではなく, 正規表現 から実行バイナリを生成することを指す (3. 2. 1 節). 本研究では, 実行バイナリの生成にはコンパイラ基盤である LLVM, GCC を用いており, 評価器全体の実装としては Python で実装した.

本論文では, まず正規表現のコンパイル方法について説明し, 実装した評価器の性能調査のために, 正規表現を用いてテキストマッチ処理を行う grep と同等の機能を実装し, GNU grep との比較を行う.

2 正規表現

2. 1 正規表現によるテキストマッチ

正規表現は与えられた文字列を受理するかどうかを判定できるパターンマッチングの機構であり, sed, grep, awk を始めとするテキスト処理ツールに広く利用されている. 正規表現には定められた文法によって記述され, 例えば, 正規表現 “$a*b$” は”a” の 0 回以上の繰り返し直後, “b” で終わる文字列 (“b” , “ab”, “$aaaab$”) を受理し, “$a(b|c)$” は “a” で始まり, 直後が “b” または “c” で終わる文字列 (“ab”, “ac”) を受理する.

2. 2 正規表現の演算

本論文では, 以下に定義された演算をサポートする表現を正規表現として扱う.

Implimentation of Regular Expression Engine with Just-In-Time Compilation.

Shinya Ryoma, 琉球大学工学部情報工学学科, Dept. of Information Engineering, Ryukyu University.

コンピュータソフトウェア, Vol.16, No.5 (1999), pp.78–83.
[研究論文] 1999 年 8 月 3 日受付.

1. 連接 二つの言語 L と M の連接 (LM) は, L に属する列を一つとり, そのあとに M に族する列を連接することによってできる列全体から成る集合である.
2. 集合和 二つの言語 L と M 集合和 ($L|M$) は, L または M(もしくはその両方) に属する列全体からなる集合である.
3. 閉包 言語 L の閉包 ($L*$) とは, L の中から有限個の列を重複を許して取り出し, それらを連接してできる列全体の集合である.

正規表現は, この 3 つの演算について閉じており, この 3 つの演算によって定義される表現は, 数学的には正則表現と定義されている. 本論文では, 特に区別のない限り, 正則表現と正規表現を同じものとして扱う.

2.3 grep

正規表現は, テキストのパターンをシンプルに記述できるという利点から, テキストファイルから, 任意のパターンにマッチするテキストを検索するなどの用途に使用される.

GNU grep は, それを実現するソフトウェアの一つであり, 引数として与えられたファイルから, 与えられた正規表現にマッチするテキストを含む行を出力する機能を持っている.

“与えられた正規表現にマッチするテキストを含む” というのは, 行の先頭から末尾まで正規表現によるマッチングを行い, 正規表現が受理状態になった時点で “含む” という解釈を行う. つまり, 正規表現 ”$(a|s)t$” は, ”at” または” st“を受理する正規表現であり, テキスト行”$math.$“の 2 3 文字目の”at” に一致するので grep は “$math.$” を出力する. また正規表現”$a*$” は, “a” の 0 回以上の繰り返しを受理する正規表現であり, 空文字も受理するので, grep は全ての行を出力することになる.

3 正規表現評価器の実装

正規表現は等価な NFA に, また NFA は等価な DFA に変換することが可能である [4]. 以下にその変換手方を説明する.

3.1 正規表現から NFA への変換

NFA(*Non-deterministic Finite Automaton*) は, 入力に対して複数の遷移先持つ状態の集合であり, 遷移先が非決定的 (Non-deterministic) である. ここでは, NFA を 5 個組 $(Q, \Sigma, , \delta, q_0, F)$ で定義する. ただし,

1. Q は状態の有限集合.
2. Σ は入力記号の有限集合.
3. q_0 は Q の要素で, 開始状態と呼ぶ.
4. F は Q の部分集合で, 受理状態と呼ぶ.
5. δ は, 状態と入力記号に対して状態の集合を返す遷移関数.(ε 遷移を許す)

正規表現が, 等価な NFA に変換できるということを,2. 2 で定義した 3 つの演算について対応する NFA に変換できることから示す.

1. 連接 図 1 は正規表現 “AB” に対応する NFA.
2. 集合和 図 2 は正規表現 “$A|B$” に対応する NFA.
3. 閉包 図 3 は正規表現 “A*” に対応する NFA.

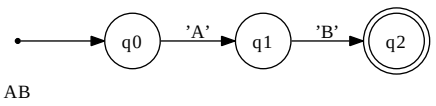

図 1 “A” と “B” の連接

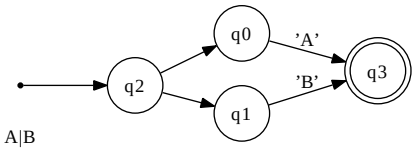

図 2 “A” と “B” の集合和

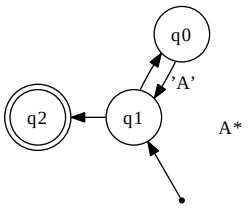

図 3 “A” の閉包

図 2, 3 において, ラベルのない矢印は無条件の遷移を現しており,ε 遷移と呼ばれる. また, 二重丸で囲まれた状態は受理状態を現しており, NFA において入力が終了した時点で, 受理状態を保持している場合に限り, その文字列を受理したことになる. なお, NFA は同時に複数の遷移先をもつことがあるので, テキストのマッチング途中で複数の状態を保持することがある.

現在実装されている正規表現評価器の多くは, 正規表現を内部的に NFA に変換して評価を行っている [1]. NFA による実装は, 後述する後方参照や最長一致に対応しやすいが, 同時に遷移する可能性のある複数の状態を保持する必要があるため, 正規表現の複雑度に多じてマッチングの時間が多くなってしまう場合がある. 文献 [1] では, "$a?a?a?aaa$" のような "$a?^n a^n$" のように表現 ("$a?$" は"a"か空文字"を認識する拡張された正規表現お一つ) の評価において, NFA ベースの正規表現評価器では遷移する状態の数が増えてしまうでマッチングにかかる処理時間が n の指数的に増加する問題をベンチマーク結果と共に指摘している. 文献 [5] では正規表現から NFA ベースで効率的なマッチング処理を行う評価器を IBM 7094 機械語で生成する例が紹介されている.

3.2 NFA から DFA への変換

非決定的な遷移を行う NFA から, 決定的な遷移を行う DFA(*Deterministic Finite Automaton*) に変換する手法を説明する. なお, 遷移が決定的であるということは, 1 つの入力に対して, 遷移する状態がただ 1 つであるということを指す. DFA は, NFA と同様な 5 個組で $(Q, \Sigma, , \delta, q_0, F)$ 定義できる. ただし,DFA において δ において ε 遷移は認められず, また任意の状態 q と入力 σ について, $\delta(q, \sigma) = q'$ となる q' は Q の要素となる. つまり, 遷移先が決定的であるということに他ならない.

以下に ε 遷移を許す ε-NFA $E = (Q_E, \Sigma, \delta_E, q_0, F_E)$ から等価な DFA $D = (Q_D, \Sigma, \delta_D, q_D, F_D)$ を構成する手順を示す.

1. Q_D は Q_E の部分集合全から成る集合であり, おの中で D において到達可能な状態は, ε 遷移に関して閉じている Q_E の部分集合 S に限られる. ここで, 状態 q において ε 遷移に関して閉じている集合全体を $ECLOSE(q)$ と表す. $ECLOSE$ を使って S を定義すると, $S = \bigcup_{q \in S} ECLOSE(q)$ を満たす S.
2. $q_D = ECLOSE(q_0)$. すなわち, E の開始状態の ε 閉包.
3. F_D は E の状態の集合で, 受理状態を少なくとも一つ含むもの全体からなる集合である. すなわち,$F_D = \{S | S \in Q_D \wedge S \cap F_E \neq \emptyset\}$
4. $\delta_D(S, a)$ は Q_D の要素 S と Σ の要素 a に対して次のように計算される.
   (a) $S = \{p_1, p_2, ..., p_k\}$ とする.
   (b) $\bigcup_{i=1}^{k} \delta(p_i, a)$ を求め, その結果を $\{r_1, r_2, ..., r_m\}$ とする.
   (c) このとき, $\delta_D(S, a) = \bigcup_{j=1}^{m} ECLOSE(r_j)$

この方法によって得られた DFA D は NFA E と同等の言語を認識し, また NFA の元となる正規表現と同等である.

3.2.1 DFA から実行バイナリの生成

DFA からの実行バイナリ生成には, 2 種類の実装を行った.

1. DFA → Continuous based C → gcc によるコンパイル
2. DFA → LLVM-API → LLVM によるコンパイル

以下, Continuous based C, LLVM それ自身の説明と, それを利用した DFA からの実行バイナリ生成の方法を説明する.

3.2.2 Continous based C

Continous based C(以下 CbC) は, ... 本研究室での先行研究により CbC コンパイラは, GNU C Compiler 上で実装されている [7], 本研究では gcc-4.5 上に実装された CbC コンパイラを用いた.

以下に, 正規表現 "$(A|B)*C$" に対応する DFA と, DFA の各状態に対応する CbC のコードセグメントの生成例を示す.

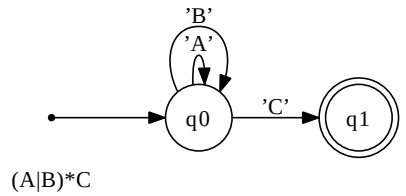

図 4 正規表現 “$(A|B)*C$” に対応する DFA

```
__code state_0(unsigned char *s, unsigned
        char* cur, unsigned char* buf, FILE
        *f, char* filename) {
  switch(*s++) {
    case 65: /* match A */
      goto state_0(s, cur, buf, f, filename);
    case 66: /* match B */
      goto state_0(s, cur, buf, f, filename);
    case 67: /* match C */
      goto state_1(s, cur, buf, f, filename);
    default: goto reject(s, cur, buf, f, filename);
  }
}
__code state_1(unsigned char *s, unsigned
        char* cur, unsigned char* buf, FILE
         *f, char* filename) {
  goto accept(s, cur, buf, f, filename);
}
```

図 4 の DFA に対応する CbC コードセグメント

DFA の遷移とは直接関係のない引数 (ファイル名やバッファへのポインタ等) が目立が, CbC では環境をコードセグメント間で引数として明示的に持ち運ぶ軽量継続を基盤としたプログラミングスタイルが望ましい. 今回コンパイラによって生成した CbC ソースコードでは, 大域変数は持たず, 必要な変数は全て引数に載せている. CbC の state_1, state_0 から呼ばれている accept, reject はそれぞれ受理状れ受理と非受理を表す. accept ではテキスト行を出力して次の行へ, reject では次の文字へと処理を移すコードセグメントへ継続を行う.

生成した CbC ソースコードを, GCC 上に実装した CbC コンパイラによってコンパイルすることで実行バイナリを得る.

3.2.3 LLVM

LLVM(Low Level Virtual Machine) は

4 評価

5 まとめと今後の課題

参 考 文 献

[1] Cox, R : Regular Expression Matching Can Be Simple And Fast, 2007
[2] Cox, R : Regular Expression Matching: the Virtual Machine Approach, 2009
[3] Cox, R : Regular Expression Matching in the Wild, 2010
[4] Hopcroft, J, E. Motowani, R. Ullman, J. : オートマトン言語理論計算論 I (第二版), pp. 39–90.
[5] Thompson, K : Regular Expression Search Algorithm, 1968
[6] 長谷川 勇 : 国際正規表現ライブラリの開発
[7] 与儀 健人 : 組込み向け言語 Continuation based C の GCC 上の実装, 2010