3L－8　部品データベースへの問い合わせをアルゴリズム記述要素として用いるプログラミング手法についての一考察*

小林弘明　岡本秀輔　曽和将容†
電気通信大学大学院情報システム学研究科‡

1　はじめに

プログラムの再利用性を改善することで，プログラムを部品として扱えるようにすることを目標とした研究 [1, 2] は数多くなされている．特にオブジェクト指向プログラミングは一定の成果を上げてきた．

これに対し，近年，知識モジュールやシステムの柔らかさと呼ばれる概念が提案されつつある [3, 4]．モジュールの柔らかさとは，モジュールの，状況変化に対する適応性の事であると考えられる．この概念は，通常言われる再利用性よりも広い概念である．すなわち，モジュール化プログラミングにおける再利用性とは，1 つのモジュールが変更抜きに他の状況下で使用できるかどうかを意味する．これに対し，柔らかさ (適応性) の場合，必要な個所を自ら書き換えるという意味を含む．

したがって，柔らかいモジュールを実現するためには，モジュールの自動修正に関する問題を扱う必要がある．図 1はモジュールの自動修正の概念図である．プログラマが外部仕様を更新した時，要求の変化に適応するように，プログラムモジュールが自動的に修正される事が本研究の最終目標である．

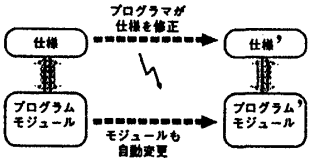

図 1: プログラム ・モジュールの自動修正

この問題の主要な困難は，モジュールがブラックボックスであることから生じる．すなわち，通常のモジュール化プログラミング言語においては，言語はモジュールの界面仕様のみを管理し，モジュールの内部に関しては関知しない．すなわち，モジュール内部の細部が担う “役割” は言語の管理下にない．このため言語の枠組の中では，モジュール全体の目的が変化した時，モジュール内部の 『どこを』 『どのように』 変更するべきかを知る事ができない．

2　アルゴリズム記述要素として部品要求を用いる理由

この問題に対する本研究の基本的アプローチを図 2 に示す．要求と最終生成プログラムとの間に中間表現を (1 段以上) 介在させる点が特徴である．自動修正は中間表現の上でなされる．

中間表現を用いることにより，最終的に生成されるプログラムの実装モデルと上位のモジュールのセマンティクスとを独立させる．これにより，(1) 上位のモジュールに関しては代数的 (等価) 変換など取扱いの容易さを保ち，かつ，(2) 抽象化階層の深さを最終プログラムの効率と独立させる．

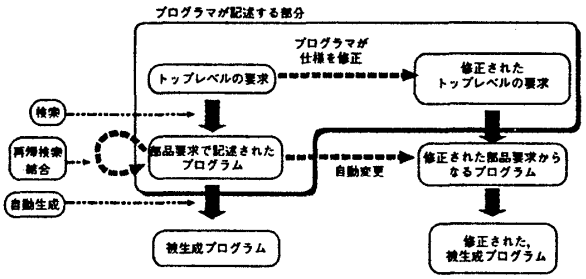

図 2: 問題に対するアプローチ

本稿で用いる主要な記法の定義は以下の通りである．

1. モジュールは，目的記述部と手段記述部の 2 つの要素からなる．目的記述部は，従来の界面仕様に相当する．手段記述部はモジュール本体に相当する．
2. 手段記述部は，前述の中間表現に相当し，アルゴリズムを可能な限り抽象的に記述するための部位である．この記述が，下請けモジュールを部品データベースから取り出すための問い合わせに変換される．
3. アルゴリズム記述を問い合わせへと変換する手順は，以下の通りである．
   (a) 記述中に出現した名前は，プログラマが別途名前ごとに定義した “問/答ペア” の集合へと置き換えられる．問/答ペアとは，その名前が指す概念の性質を表すためのもので，質問と，それに対する答 (True/False/don't care) のペアである．
   (b) 連続する問/答ペア集合は，集合の包含関係に基づき，左から順に単一化される．
   (c) 中置き演算子 “:” は，IMPLY 演算であり，規則 3b よりも先に適用される．
4. 同じ手順を用いて，モジュールの目的記述部を問/答ペア集合へと変換し，このモジュールが検索される時のキーとして用いる．

3　提案手法の適用例

本研究は端緒期にあるため，動作可能な実例は未だ存在しない．そこで，妥当性の検証として，提案手法の全体的な枠組を明らかにし得る具体的なプログラムについて，修正を行う例を挙げる．

図 3 は修正前の，オリジナルプログラムに相当する，C 言語によるプログラムである．改行により終端された文字列を 1 行と数えている．このプログラムに対して修正を加える事で，コメント記号ではじまる行を数えないようにすることを考える．図 4は，そのような修正によって得られるべき目標プログラムの一例である．

---

*A Study on “Module Query as Atom of Algorithm”.
†Hiroaki KOBAYASI,Shusuke OKAMOTO,Masahiro SOWA
‡Graduate School of Information Systems, University of Electro-Communications, Tokyo, 182 Japan

```
int linecount()
{
  int count = 0;
  while((c = getchar()) != EOF){
    if(c == '\n'){
      /* '\n' を見付けたら */
      count++;
      /* カウンタの値を増やす */
    }
  }
  return count;
}
```

図 3: 元のプログラム…行数勘定プログラム

```
int linecount_ignorecomment()
{
  int count, c, last;
  last = 0;
  count = 0;
  while( (c = getchar()) != EOF ){
    if((last == 0 || last == '\n') && c == '#'){
      /* "'#['\n]*'\n" を読み飛ばす */
      while( (c = getchar()) != EOF ){
        if(c == '\n') {
          break;
        }
      }
    } else if(c == '\n'){
      count++;
    }
    last = c;
  }
  return count;
}
```

図 4: 修正によって得たい, 目標プログラム

提案する手法において, プログラマは図3のプログラムを複数の部品プログラムに分割して記述する. 図5 は図3のプログラムを生成するために必要な部品プログラム群の一部を表す図である.

```
=== 部品1 ===
目的: [count per: line from: stdin]
手段: [line recognition] ⇒* [count]

=== 部品2 ===
目的: [line recognition]
手段:
 {[iterate per: [character "C"] from: stdin]
   [character "C"] ==
    (
      ¬[line terminator] ⇒ [element of: line]
       [line terminator] ⇒ [line match]
    )
 }

=== 部品3 ===
目的: [count per: unit]
手段: {[iterate per: character from: stdin]
       BEGIN          ⇒ [initialize [storage "X"]]
       [unit match]   ⇒ [inc [storage "X"]]
       END            ⇒ [return [value of: [storage "X"]]]
      }

=== 部品4 ===
目的: [[line for: comment] recognition]
手段:
 {[iterate per: [character "C"] from: stdin]
   [character "C"] ==
    ([beginning of: line] & [comment leader]
          ⇒ ([line recognition]
               ⇒ [[line for: comment] match]
              )
    )
 }

=== 部品5 ===
目的: [["X" - "Y"] recognition]
手段: (["Y" recognition] ⇒ [skip]
       ELSE ⇒ ["X" recognition]
    )
```

図 5: データベース内の部品群 (一部)

本手法においては, プログラマは, 部品データベースとともに, 部品に出現する名前について, その名前が表す概念の性質に関する記述 (問/答ペア集合) を与える事を要求される. 部品検索の際の一致判定は, この性質記述同士の包含関係に基づいて行われる. 図6は性質記述のごく一部の例である. この例は, line という名前の指す概念について, それが関数としては利用できず (function=F), 単位として利用できる, という事を表している.

```
line  ::= { function=F, unit=T, ···}
count ::= { function=T, unit=F, ···}
per   ::= ···
```

図 6: 名前と問/答ペア集合の対応表

図 7はプログラマが与える修正指示の書式と, その例である. 一般的なプログラミング言語におけるレコード (構造体) 記法と同様に, 『対象』, 『修正部分』, 『修正内容』 の3つ組からなる.

```
書式: [モジュール要求] ·[修正個所]=[修正方法]
例:   [count per: line] ·[line] =[line − [line for: comment]]
```

図 7: プログラマが与える修正指示の例

図 8は, 図7の修正指示を図5に与えた時に自動生成されるべき部品の一覧である. ここで行われた書き換えは, トップレベルのモジュール (部品 1) における line という名前を [line − [line for: comment]] へと書き換えたこと, およびそれによって誘導される書き換えを順次適用したこと, という単純なものである. このため, 少なくともこの例に関しては, 書き換えは自動化可能であると考えられる.

```
=== 部品1' ===
目的: [count per: [line − [line for: comment]] from: stdin ]
手段: [[line − [line for: comment]] recognition] ⇒* [count]

=== 部品2' ←(部品2+ 部品5)===
目的: [[line − [line for: comment]] recognition]
手段:
 {[iterate per: [character "C"] from: stdin]
   [character "C"] ==
    (
      [[line for: comment] recognition] ⇒ [skip]
      ELSE ⇒ [line recognition]
    )
 }
```

図 8: 修正により得られるモジュール

4 おわりに

本稿では, アルゴリズム記述要素として部品データベースへの問い合わせを用いる手法について述べた. この手法の全体的な枠組と, それが "柔らかい"(=自動修正を行いやすい) モジュールを記述する方法として利用できる可能性を示す例を挙げた.

参考文献

[1] Bertrand Meyer: Object–Oriented Software Construction. Hertfordshire, England: Prentice Hall International, (1988).

[2] Futatugi, K. Goguen, J. Jouannaud, J.–P., and Meseguer, J.: Principles of OBJ2, Proc. of 12th Symposium on Principles of Programming Languages, ACM, pp.52–66.(1985),

[3] 蓬莱 尚幸: モジュール間協調に基づく軟らかい形式的仕様記述, 第 47 回 情報処理学会 全国大会論文集, 第 5 分冊,pp25-26,(1993).

[4] 白鳥則郎, 菅原研次: やわらかいネットワークの開発に向けて — 知識型設計方法論 — , 電子情報通信学会 技術研究報告,AI93–46, (1993).

[5] 小林弘明, 有田隆也, 川口喜三男, 曽和将容: 概念制約式を用いたプログラミングとプログラム合成, 第47回 情報処理学会 全国大会論文集, 第 5 分冊,pp19-20,(1993).