ソフトウェア工学　52 — 5
（１９８７・２・１３）

論理的な解析にもとづく
関数型プログラミングの支援法

鈴木賢三　　　永田守男
慶応大学理工学部管理工学科

プログラムの動的なテストは誤りの発見には有効だがプログラムの正しさを保証するものではなく、また実行可能なプログラムになっていない段階のものはテストできないので、段階的詳細化におけるプログラミングとテストの時間的なずれは解消できない。
本報告ではプログラムの正しさを確かめることと並行して段階的にプログラムを詳細化する方法を提案し、この方法に沿ってプログラミングを支援するシステムの概要を述べる。本方法では、実現されていない部分を含んだプログラムと形式的仕様から派生する論理式の書き換えを繰返し、未実現の部分に関する条件を生成し、この条件をその後のプログラミングの指針とすることができる。対象は関数型の言語で、関数とデータ型を階層的に具体化する。この方法では、概要レベルの正しさを確かめてから詳細レベルのプログラミングへ進み、最終的に論理的な正しさが保証されたプログラムが得られる。また誤りが見つかってもその修正による他の部分への影響が少なく、信頼性の高いプログラミングが期待できる。

A SUPPORT METHOD FOR FUNCTIONAL PROGRAMMING BY LOGICAL ANALYSIS

Kenzo Suzuki and Morio Nagata

Dept. of Administration Engineering ,
Faculty of Science and Technology , Keio University
3-14-1 , Hiyosi , Yokohama , 223 JAPAN

Program testing is useful for indicating errors in the program. However, it can not confirm the correctness of the program. Furthermore, we can test only executable programs. Thus testing is not usable for all steps of program developing in stepwise refinement.
This paper proposes a method for verifying the correctness of programs during all steps of the program development in stepwise refinement. We show a programming support system based on this method.
In our method, a programmer writes both an incompleted program and logical expressions specifying its function in every step of the development. The support system confirms the correctness of the program. Moreover, it generates some conditions to specify some incompleted parts of the program. The programmer repeates this process until the program is completed.
In this study, we use a programming language in the functional style. We refine both functions and data during the steps of the program development. If a progarmmer uses our system and writes his/her program in accordance with our method, he/she can writes a completed program those correctness is verified. When the programmer detects an error, he/she can correct it without worrying about many parts of the program.

1．はじめに

信頼性の高いプログラム作成に関する議論は、一般にプログラミングの方法論と記述の済んだプログラムの正当性確認の2つに分けて考えられる。

構造化プログラミング〔1〕や抽象データ型〔14〕の導入などは、理解しやすく変更しやすいプログラムを書くためには優れた方法だが、こうして書かれたプログラムも、やはり実際にいくつかのテストデータを通すなどしてその正しさを確かめなければならない。

プログラムの正当性確認の方法としては、テスト、記号実行、プログラムの検証などがある。

プログラムのテストは最も一般的な正当性確認の方法であり、そのためにデバッガなども作られ、誤りの検出などに使われている。しかしプログラムのテストは、誤りがないこと、すなわち正当性を保証してくれない。

記号実行〔2,3〕は、実際のデータでプログラムを実行する代りに一般的なデータを代表する記号をプログラムの入力としてプログラムを実行する方法である。これはプログラム変数の値を記号式として表現し、通常の実行を自然に拡張したものといえる。

プログラムの検証については後述するが、ここで述べた二つの手法に共通していることは、すでに詳細なレベルまで完全に記述の済んだプログラムを対象としている点である。

段階的詳細化法〔4〕のように、プログラムが概要レベルから詳細レベルへとトップダウンに作成されるのが自然なのに対して、プログラムの正当性確認はモジュールごとに正しさを確かめ、それらの積み重ねとして全体の正しさを確かめるというように逆の方向でなされることが多い。テストによって正当性確認をする限り、プログラム作成と正当性確認が別々な作業として扱われ、この2つの時間的なずれは解消できない。

こうした問題をふまえ、本稿では、プログラムの検証の考え方をもとに、プログラム記述と論理式の書き換えを繰返しながら、プログラムを正しく詳細化する方法を提案し、いくつかの例題についてこの方法の適用を試みる。さらにこの方法に基づいて設計したプログラミング支援システムについて説明する。

2．プログラミングと検証の統合

プログラムの動的なテストと相対する方法としてプログラムの検証が考えられている。これはプログラムとプログラマの意図を表す形式的な仕様から、プログラムの正しさを論理的に証明するものである。形式的な仕様の記述などの手間はかかるが、正しさが証明されればプログラムに誤りがないことが保証される。証明を機械的に行なわせるプログラム検証システム〔5〕も存在するが、次のような理由から現実のプログラムでは形式的な検証が難しいと考えられている〔6〕。

1. 証明に必要な帰納的表明の発見が困難である。
2. 形式的仕様の記述が困難である。
3. 十分な記述能力をもった仕様記述言語がない。
4. 定理証明の能力が不十分である。
5. 証明過程であまりに詳細なレベルまでプログラマの支援を要する。

さらに、プログラム検証の最も根本的な問題として正しいプログラムの”正しさ”だけを保証している点があげられる。検証に失敗した場合、検証システムはプログラムあるいは形式的仕様記述の中のどこに誤りがあるかを教えてくれない。

検証が誤りの発見に役立たないというのは、主に検証を機械的な定理証明と考える傾向が強いためといえる。証明の過程を人間がトレースすればどこで証明が行きづまったのか、そしてどのようにプログラムや仕様を直せば証明がうまくいくかを知ることができよう。またプログラムの検証は、動的なテストと異なり、実現されていない部分を含むプログラムにも適用可能である。

本研究ではこうした観点から、形式的な仕様をもとに抽象度の高いプログラムを論理的に解析し、ここの結果を誤りの修正に利用し、正しく記述されたことを確かめながら詳細化を進め正しいプログラムを作成する方法を提案する。あわせて上に掲げたような問題点について検討を加え、プログラム検証を実用的な水準に近づけるための方法について考察してみる。

3．記述言語

実際のプログラミングについて考えるためには言語を特定しなければならない。ここでは論理的な扱いが容易なこと、自然に階層的なプログラミングができること、またデータの流れに着目してプログラミングを考えることができることなどの理由から関数型のプログラミングを対象とした。

ここでは主に簡単な数値計算とリスト処理を扱うことにするが、ＬＩＳＰのような既存の言語では機能が多すぎてカバーしきれないため、プログラミングに支障のない範囲で次のような制限を加えた言語を設計した。

1．副作用をもたない
2．高階関数を使用しない
3．繰返しは再帰だけを用いて記述する

また、ＬＩＳＰと異なる点として全ての引数と関数値のデータ型の宣言を導入した。言語の性格上リストに関する処理が中心となるが、単に”リスト”という型宣言を用いて関数が定義された場合でも、プログラマが意図している処理対象は全てのリストではなく、ある特定の構造を持ったリストを対象と考えているかもしれない。そこでこのような特定のリストに対するデータ型の名前を定義してその構造を階層的に具体化していくことができるようにした。こうした型の導入は定義した関数の意図を明確にするとともに、関数の呼び出しにおける不整合を静的にチェックすることを狙っているが、検証においても、本来起こりえないようなケースまで考えずにすむため効率の良い検証が可能となる。

3．1 関数の記述

言語仕様は関数型言語Valid〔13〕をもとに設計した。関数を定義するときの仕様の一部をＢＮＦ風に記述したものを図1に示す。ここで（ ,）は入れ子を、 A1|.....|AnはA1からAnのうちいずれかひとつの構造をもつことを意味する。また＊は0個以上の、＋は1個以上の有限な並びを、〔 , 〕は省略可能をあらわす。

〈関数定義〉の最初の2行は入出力値の型宣言である。〈出力タイプ〉は複数個の型名を並べて書くこと

ができ、この場合それぞれの型の値をリストにして返す。〈出力タイプ〉が1個の場合は、値をそのまま返す。〈式〉は必ずそれと対応する1個以上のreturnをもち、returnで与えられた値を返す。let,whereは内部パラメータを定義する。その定義は逐次的で〈等式〉の並びの中で既に定義されているパラメータは以下の〈等式〉で参照できる。if文は一般に用いられるものと同じである。case文はＬＩＳＰのcondに似ており、〈述語〉の箇所に陽に真理値を書くこともできる。全ての〈述語〉が偽の場合、caseは nilを返す。

```
〈関数定義〉
   ::= (function 〈関数名〉 ( 〈入力タイプ〉* )
         return          ( 〈出力タイプ〉+ )
         by
        〈関数名〉 ( 〈S式〉 ) == 〈式〉 )

〈式〉
   ::=  (let ( 〈等式〉* ) 〈式〉 )
      | (return 〈S式〉 [where ( 〈等式〉* )])
      | (if ( 〈述語〉 ) then 〈式〉 else 〈式〉)
      | (case (( 〈述語〉 ) 〈式〉 )

〈等式〉
   ::=  〈S式〉     = 〈関数名〉 ( 〈引数名〉* )
      | 〈S式〉     = 〈アトム〉
      | 〈アトム〉 = 〈アトム〉 〈演算子〉 〈アトム〉
```

図　1　　関数の記法

3．2 データ型の記述

この言語はＬＩＳＰと同様にＳ式という唯一のデータ型を対象としており、これは図2に示したようないくつかの基本データ型によって構成されている。

ＬＩＳＰでは真理値のfalseと空リスト（ ）を同じシンボルnil で表現しているがここではそうした曖昧さを避けた。ブールは *true* と *false*の2つからなり他のアトムでは”＊”などの特殊記号を使えない。（例外としてハイフンとアンダーバーを許す）またこれらのデータ型をもとにプログラマはある特定の構造をもったデータ型を定義して利用できる。データ型の定義には次の5つのパターンがある。

1．(define-data-type A by A1 A2 .....An)
　A およびAi (1≦i≦n)はいずれも型の名前。
　データ型AはAiの和集合として定義される。
2．(define-data-type A by (* B))
　A,Bは型の名前。AはBの0個以上の（有限個の）集合として定義される。
3．(define-data-type A by (+ B))
　AはBの一個以上の（有限個の）集合として定義される。

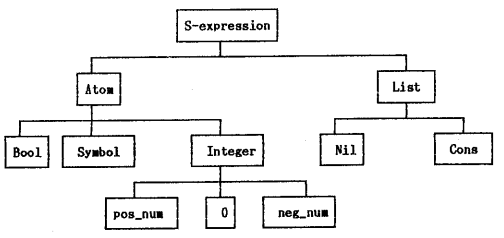

図　2　　基本データ型の階層

4．(define-data-type A by (S1 S2 ....Sn))
　Si(1≦i≦n)はシンボルまたは型の名前。（シンボルを型の名前と区別するため ’を用いる）
　データ型Aはn個のS式S1からSnまでを順に並べたリストとして定義する。
5．(define-data-type A by B)
　AはBと同一の型として定義される。1．の特種なケース。

3．3　仕様の記述

形式的仕様記述の難しさは、プログラムの条件を一度に厳密に記述しようとするために生じる。本研究では仕様を関数の入出力値に関する性質を表現した述語、あるいは述語の集合と考える。述語は関数値としてブールを返すという性質を持った関数の一種と見なすことができる。従って述語は関数と同様に詳細な定義を与えることにより、その意味が明確に記述できる。しかし述語の厳密な定義は必ずしも必要なく、それに関する補助定理を適宜与えるなどして済む場合が多い。

例．二つの整数の最大公約数を求める関数ｆを定義する。ｆの仕様を与える場合、まず
「x,yの最大公約数はｚである」　　(3.1)
ことを形式的に表現しなければならないがこれは容易なことではない。
そこでまず、(3.1)を
　　gcd(x,y,z)
と書くことにする。
ｆが正しく最大公約数を計算していることをしめすには
　　gcd(x,y,f(x,y))　　(3.2)
を証明すればよいが、そのためには(1)を具体的な表現に置き換えねばならない。
「ｚがｘ，ｙの最大公約数である。」とは
「ｘ，ｙがともにｚで割り切れ」　　(3.3)
かつ
「ｘ，ｙを数ｚで割った商はたがいに素である」　　(3.4)
といえる。(3.3),(3.4)に相当する述語を各々ｐ1，ｐ2とすると
　gcd(x,y,z)≡p1(x,y,z)∧p2(x,y,z)
と書ける。さらに、「aがｂの倍数である」ことを表す述語を
　multiple(a,b)
「aをｂで割った商」を表す関数を
　quotient(a,b)
「a，ｂはがいに素である」ことを表す述語を
　prime(a,b)
とすれば
　p1(x,y,z)≡multiple(x,z)∧multiple(y,z)
　p2(x,y,z)≡prime(quotient(x,z),
　　quotient(y,z))
と書ける。こうして得られた関数や述語の定義あるいは補助定理を与えて証明をすすめる。
厳密には(3.2)は
　f:nat,nat　→ nat　　(3.5)
gcd:nat,nat,nat　→ bool　　(3.6)
((3.5)のような記法は、ｆが二つの自然数を受け取り一つの自然数を返す関数であることを意味する。以下同様)
のもとで
　∀x∈nat ∀y∈nat gcd(x,y,f(x,y))
を主張している。論理式中の∀変数は定義域内の任意の値を取るものと考え、式の中に陽に全称記号や存在記号は書けない。このような形式での仕様記述は一階の述語論理より弱いが、実用上の障害はほとんどない。

4．例題

形式的仕様をもとにプログラムを解析する過程を簡単な例を用いて説明する。

例題1．整数のリストを受け取り泡立ち法によってソートする関数bubble-sortを定義する。

```
(function-definition bubble-sort (int-list)
          return                 (int-list)
    by
 bubble-sort (list_0) ==
   if list=nil
   then nil
   else (let
          ( (int_0.list_1)= select-min(list_0)
              list_2      = bubble-sort(list_1))
          (cons int_0 list_2)))
(define-data-type int-list
    by
  (* integer))
undefined function
     select-min : int-list → int-list
```

ソーティングについては次のような性質が成り立たねばならない。

性質1：リスト内の要素がソートされている。
性質2：ソーティングの前後でリストの構成要素が同じ

そこでまず性質1に関する論理式を考えてみる。int-listを受け取りソート済みか否かを真理値で返す述語をsortedとする。

```
  sorted : int-list → bool
```

関数bubble-sort について性質1が成り立たねばならないから

```
  (sorted (bubble-sort l))                (4.1)
```

が正しいはずである。そこでこの式を書き換えながら未定義の関数select-minや述語sortedについて考えてみる。

(4.1)は次のように書き換えられる。

```
  x=(bubble-sort l) → (sorted x)      (4.2)
```

bubble-sortの定義よりlがnilの場合とそうでない場合とにわける。

```
  〔l= nil ⊃ x=nil〕 → (sorted x)          (4.3)
  〔l≠nil ⊃ x=(cons
               (car (select-min l))
               (bubble-sort
                 (cdr(select-max l)))))〕
                → (sorted x)          (4.4)
```

(4.3)より

```
      (sorted nil)
```

すなわち述語sortedはnilについて成り立たねばならない。

(4.4)に帰納法を適用して整理すると

```
  〔l≠nil ⊃ 〔m=(select-min l)
           ⊃ 〔n=(bubble-sort (cdr m))
            ⊃ 〔(sorted n)
             ⊃ x=(cons (car m) n) 〕〕〕〕
               → (sorted x)              (4.6)
```

ここで述語sortedについて考えてみる

```
  ∀a∈integer b∈int-list
    〔(sorted b) ⊃ x=(cons a b)〕→(sorted x)
                                        (4.7)
```

は正しいだろうか。これは明らかに誤りで成立するのはbのすべての要素がaより大きいか等しい場合に限られる。このことを表す述語minimumを用いて(4.7)を修正し

```
  〔(minimum a b)∧(sorted b) → x=(cons a b)〕
          → (sorted x)                  (4.8)
```

(4.8)を利用して(4.6)を書き換えると

```
  〔l≠nil ⊃ 〔m=(select-min l)
            ⊃ n=(bubble-sort (cdr m)) 〕〕
             → (minimum (car m) n)      (4.9)
```

ここで性質2をすでに証明済みとして利用する。二つのリストの構成要素が一致するか否かを答える述語をset-equalとする。bubble-sortについてset-equalが成り立つことは

```
   (set-equal x (bubble-sort x))          (4.10)
```

と書ける。これを利用すると(4.9)は

```
  〔l≠nil⊃〔m=(select-min l)
           ⊃〔n=(bubble-sort (cdr m)
             ⊃(set-equal n (cdr m)) 〕〕〕
               →(minimum (car m) n)
```

さらに補助定理

```
 (set-equal a b)∧(minimum x a)
              → (minimum x b)
```

を適用し整理して

```
 〔l≠nil → m=(select-min l)〕
             → (minimum (car m) (cdr m))
```

次に性質2についてつまり(4.10)を証明する。まずthen節より

```
 (set-equal nil nil)
```

else節より

```
 〔x≠nil ⊃ 〔m=(select-min x)
           ⊃ 〔n=(bubble-sort (cdr m))
             ⊃ y=(cons (car m) n) 〕〕〕
               → (set-equal x y)
```

帰納法および2つの補助定理

```
   (set-equal x (cdr y))
     → (set-equal (cons (car y) x) y)
   (set-equal a b)∧(set-equal b c)
     → (set-equal a c)
```

を適用すると、

```
   x≠nil ∧ m=(select-min x)
     → (set-equal x m)
```

となる。以上をまとめると、未定義の関数select-minが次の二つの制約条件

```
  x≠nil ∧ m=(select-min x)
    → (minimum (car m) (cdr m))
  x≠nil ∧ m=(select-min x)
    → (set-equal x m)
```

を満たすように定義されれば関数 bubble-sortは正しい。

例題2．与えられたリストの内容を反転させる関数reverseを定義する。

```
(function-definition reverse (list)
              return          (list)
  by
 reverse (x) ==
      if    (null x)
      then  nil
      else  (append (reverse (cdr x))
                    (cons (car x) nil)))
```

ここで、reverseの前後でリストの長さが変化しないこと、つまり

```
(length x) = (length (reverse x))    (4.12)
```

を証明したい。ただしリストの長さを求める関数をlengthとする。証明したい論理式は、以下のように書ける。

```
(listp x) ∧ y = (reverse x)
        → (length x) = (length y)
```

この式を2つの補助定理

```
(listp x) ∧ (listp y) →
        (length (append x y))
           = (length x) + (length y)
(listp x) →
        (length (cons x nil)) = 1
```

を用いてreverseとyが排除されるように書き換えていくと以下のようになる。（途中略）

```
(listp x) ∧ ¬(null x)
     → (length x) = (length (cdr x)) + 1
                                    (4.13)
```

ここで(4.13)に注目すると，後件はxがnilでない場合のlengthの定義を与えてくれている。これをもとにlengthは次のように定義できる。

```
(function-definition length (list)
             return          (integer)
    by
  length (x) ==
       if   (null x)
       then 0
       else (length (cdr x)) + 1 )
```

このlengthの定義が意図したとおりのものならば、reverseは(4.12)を満たす。

5．論理式の書き換え

論理式は原則としてシーケント(sequent)を拡張して

〔環境〕　仮定　⇒　結論

の形で表現され、自然演繹に関する公理と推論規則に基づいて書き換えられる。論理式中に未定義の関数が含まれる場合、論理式の真偽が決まらない可能性が強い。この場合、既定義の関数を排除して未定義の関数の制約条件を生成する。そのためには次のような規則が考えられる。

a．仮言命題の作成

関数fの形式的仕様が述語Rを用いて

```
R(x,y,f(x,y))                        (5.1)
```

と与えられたとき、(5.1)を後件とするシークエンスを作成する（これを仮言命題という）。この場合前件は空だが環境としては仕様中に含まれる全ての変数に関する型宣言が加わる。

```
f : t1,t2 → t3                       (5.2)
```

であれば

```
〔x∈t1,y∈t2〕 ⇒ R(x,y,f(x,y))        (5.3)
```

となる。

b．パラメータの導入

論理式を見やすくするため、パラメータを用いて等価な式に書き換える。たとえば(5.3)は

```
〔x∈t1,y∈t2〕 ⇒ z=f(x,y) ⊃ R(x,y,z)
```

と等価であり、さらに⊃-導入 の規則より

```
〔x∈t1,y∈t2〕 z=f(x,y) ⇒ R(x,y,z)    (5.4)
```

と書き換えられる。

c．関数の定義に基づく展開

c-1．if節の展開

```
f(x,y) == (if p(x)
             then g(y)
             else f(h(x),y))         (5.5)
```

とすると、(5.4)は二つの仮言命題

```
〔x∈t1,y∈t2〕p(x)⊃z=g(y) ⇒ R(x,y,z)  (5.6)
〔x∈t1,y∈t2〕¬p(x)⊃z=f(h(x),y) ⇒ R(x,y,z)
                                     (5.7)
```

に書き換えられる。

c-2．let,where,caseを含む式の展開

fが単純にif文で定義されているときは、そのままthen節とelse節に分けて考えられるが、letやwhereによって内部変数を使用している場合は以下のようになる。

letに関しては次のように書き換えられる。

```
z=(let () 〈式〉 )          ⇒          z=〈式〉
z=(let ((等式)) 〈式〉) ⇒ 〈等式〉⊃z=〈式〉
z=(let ((等式1)〈等式2)...〈等式n)) 〈式〉)
     ⇒ 〈等式1〉
          ⊃z=(let ((等式2)...〈等式n)) 〈式〉)
```

whereは次式により let形式に書き換えられる。

```
z=(return 〈S式〉 where ((等式)) ))
     ⇒ z=(let ((等式)) (return 〈S式)))
```

case形式は次の規則によりif形式に書き換えられる

```
z=(case) ⇒ z=nil
z=(case ((等式)) 〈式〉)
     ⇒ z= (if ((述語)) then 〈式〉 else nil)
z=(case (((述語1)) 〈式1))
                :
          (((述語n)) 〈式n)))
```

```
=) z= (if ((述語1)) then (式1)
        else  (case (((述語2)) (式2))
                    :
              (((述語n)) (式n))))
```

d．帰納法の適用

(5.6) はfを含まないのでgあるいはpの制約式として利用できる。一方(5.7) はfを含んでおり、fの定義をもとに論理式の書き換えを繰返してもelse部のfが残ってしまう。そこで(5.7) では f(h(x),y)に関しては既に性質Rが成り立っていることを仮定として加える。その結果 (5.7)は次のように書き換えられる。

```
〔x∈t1,y∈t2〕
        ¬p(x) ⊃
        〔R(x,y,f(h(x),y))⊃z=f(h(x),y)〕
          =) R(x,y,z)                    (5.8)
```

e．補助定理の適用

```
S(x,y,z) =) R(x,y,z)
```

が補助定理として成り立てば、(5.8)は

```
〔x∈t1,y∈t2〕
        ¬p(x) ⊃
        〔R(x,y,f(h(x),y))⊃z=f(h(x),y)〕
          =) S(x,y,z)
```

と書き換えられる。

複数の性質の成立を仮定する場合、たとえば性質1の証明において他の性質2の成立を定理として利用してもよい。ただしそのとき性質2の証明に性質1は利用できない。(bubble-sortの例では性質1の証明で性質2を利用している。)

f．代入の規則

これはパラメータの導入と逆の操作である。仮言命題の前件中に出現する変数zに関する等式の右辺が定数で与えられている場合、命題の中の全ての束縛されたzの出現を定数で置き換え、この等式を削除する。たとえば

```
〔x∈t1,y∈t2〕 p(x)∧z=nil =) R(x,y,z)
```

は

```
〔x∈t1,y∈t2〕 p(x)          =) R(x,y,nil)
```

と置き換えられる。

6．プログラミング支援システムＦＰＳ

これまでに述べた方法によってプログラミングや論理式の書き換えを円滑に進めるために現在開発中のツールＦＰＳ(Functional Programing Support System by Logical Analysis)について述べる。ＦＰＳはMicro VAX上のVAXLISPで書かれている。

6．1 ＦＰＳの概要

ＦＰＳにはつぎのような機能をもつ。

1. ユーザの入力した関数、データ型、形式的仕様の構文上の正しさをチェックし管理する。
2. ユーザの必要な情報を提供する。
3. 論理式の書き換えを支援する。
4. 記述の済んだプログラムを実行可能な形式に変換する。

これらを実現するためＦＰＳは図３のような構成になっている。

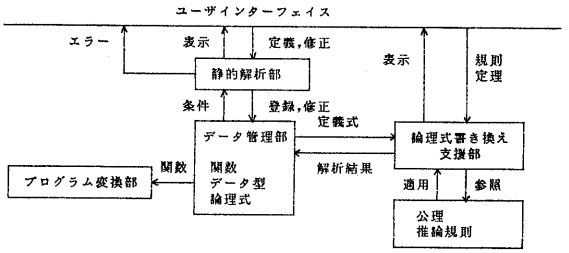

図　３　　ＦＰＳの構成

6．2　静的解析

静的解析部は、構文チェック、型チェック、バージョン管理の３つの処理をする。構文チェックは、ユーザの入力に構文上の誤りがないことをチェックする。関数などの定義は、ファイルから読み込む方法のほか、簡単なエディタを利用した入力が可能だが後者は主としていったんファイルから読み込ませたものの修正に利用する。このシステムでは、関数やデータ型の入力の順序を制限していない。

型チェックは関数同士のデータ依存関係に矛盾が無いことを確かめる。たとえば、

```
f(g(x))
```

という表現があった場合

```
f:t1 → t2
g:t3 → t4
```

として２つのデータ型t1,t4について

```
t4 ⊆ t1                                  (6.1)
```

が成り立たねばならない。t1,t4 はこうした制約を満たすように定義される。

今のところ、この程度の単純な型チェックは可能だが、たとえば

```
(if p(x) then f(g(x)) else .....)
```

のような表現が与えられた場合

```
p(x) ⊃ f(g(x))
```

つまり、p(x)が成り立つ場合だけに限って(6.1)が成立すればよくt1,t4の全域的な包含関係ではない場合話は複雑になる。この点についてはさらに考察を要する。

仕様が記述され、その正しさが証明されたり、関数の制約条件が生成されても、もとの定義が変わってしまえば、これらは意味が無くなる。こうした状況に応じて、関数や仕様を管理するのがバージョン管理である。

6．3　論理式書き換えの支援

論理式は述語論理に関する書き換え規則、5で述べたようなこの種の問題特有の書き換え規則、あるいは補助定理の適用などによって書き換えられる。論理式は最終的にはその正しさが証明されなければならないが、通常は未定義の関数を含んでいる場合が多く、ここでの書き換えは未定義の関数の制約条件を得ることが目的である。

したがって証明を目的とした式の書き換えの一過程という以上に見やすく利用しやすい式への変換が必要で、この点で一般の定理証明システムと異なる。論理式の書き換えは試行錯誤的な要素を含むためほとんど対話的に行われ必要があればバックトラックもできる。

書き換え規則は内部では条件部と実行部からなるルールの形で記述されておりパターンマッチングによって書き換える。ルールの記述方法はＯＰＳ５〔7〕を参考にした。ルールと対応する推論規則の例を図４に示す。

```
not-left
   if    (== (left)  (not (expression) ))
   then  (delete-expression (left) )
         (insert-right (expression) )

rule-8
   if    (== (left-1)  (type 'list  (x))
         (== (left-2)  (not (null (x) )))
   then  (delete-expression (left-1) )
         (delete-expression (left-2) )
         (insert-left  (type 'cons  (x)))
```

not-左

$$\frac{\Delta \rightarrow \Gamma,\ E}{(\text{not } E),\ \Delta \rightarrow \Gamma}$$

rule-8

$$\frac{(\text{type 'cons } x),\Delta \rightarrow \Gamma}{(\text{type 'list } x),(\text{not } (\text{null } x)),\Delta \rightarrow \Gamma}$$

図 4　　ルールの例

このほか補助定理の生成支援法について現在検討中である。例題でも見てきたように論理式の書き換えのポイントは適当な補助定理の発見である。ＦＰＳでは書き換え途中の論理式をもとに補助定理となりそうな論理式を生成してユーザーに提示する。補助定理が発見できないのは、主として、論理式が繁雑になり過ぎて、一般的な性質が人間に見えにくくなるためであり、システム側では、式の一部分を抽象的にとらえ、対話形式で、使えそうな補助定理の生成を試みる。

例．仮言命題が

$$\{x\in t1, y\in t2, z\in t3\}\ p(x,y,z) \rightarrow f(g(x,y))=h(z) \qquad (6.2)$$

のとき、x,yの変域や、gの値域を全く考えずに、fの定義域の任意の値wについて

$$f(w) = h(z) \qquad (6.3)$$

の成否を考える。もし正しくない場合(6.3)を成立させるための制約gを与え、補助定理

$$g(w,z) \rightarrow f(w) = h(z)$$

を生成すれば(6.2)は新たな仮言命題

$$\{x\in t1, y\in t2, z\in t3\}\ w=g(x,y)\wedge g(w,z) \rightarrow p(x,y,z)$$

に置き換えられる。

7．検討

これまで述べたプログラミングの方法及びシステムＦＰＳについて検討してみる。

1．本研究では「繰返しは全て再帰で記述する」という制限をつけたが、これはループでの帰納的表明の発見を回避する手段である。手続き型言語の検証法である帰納的表明法では、入出力に関する表明のほかに１つの繰返しについて最低１つの表明を必要とするが、この表明の発見が困難なことが多いためである。しかし関数型のプログラミングに慣れていない人にとって再帰的な記述は難しいと思われる。

2．再帰的な記述という点でこの言語は関数型言語に習熟した人向きと述べたが、仕様記述についても同様のことが言える。プログラム中で述語を使ったり、新たな述語を定義することに慣れていれば、ここで述べたような仕様記述はそれほど困難ではないだろう。

3．「定理証明の能力が不十分である」とは、実行時間と記憶領域に関する問題と、推論過程に関する問題とに分けられる。ＦＰＳは自動検証システムではないので実行時間や記憶領域の問題はないが、今後、部分的な自動化を考えた場合、なるべく抽象度の高いレベルで証明を進めるため、むやみに探索空間が広がる可能性は低い。また、データ型を明確に定義していれば、効率の良い検証ができるだろう。推論過程については、簡単な例題でも相当複雑な補助定理が必要なことがわかった。補助定理の生成方法についてさらに考えていく必要がある。

8．おわりに

本稿では未定義の部分を含むプログラムの論理的な解析を通じて、正当性の確認と並行して、プログラムを詳細化する方法について述べた。

本研究と同様に関数定義を対象とした Boyer & Moore の証明システム〔8〕がある。これはバックトラックを用いず、ヒューリスティックスの適用を含む一定のアルゴリズムに従って証明を進めるが、記述が完了していることを前提としている。

福島らは、検証と並行したトップダウンなプログラムの作成法を提案している〔9〕。この場合詳細レベルの仕様記述が必要だが、検証に関してはかなりの自動化が望める。本稿で述べた方法では、詳細レベルの仕様記述を不要とした分だけ解析に負担がかかっている。

ＨＩＳＰ〔10,11〕では、代数的仕様を基礎とし、仕様の変換によって、プログラムを導出する。また与えられた問題と等価な定理を考え、その証明を通じてプログラムを作成する方法〔12〕も考えられている。

本研究の視点はこれらと類似しているが、既存の言語によるプログラミングの経験者が、少しでも抵抗なく利用できることをねらいとしている。

参考文献

〔1〕 E.W.Dijkstra,et al., "Structured Programming", Academic Press London,INC. (1972)
〔2〕 J.C.King, "Symbolic Execution and Progarming Testing", CACM, Vol.19, No.7 (1976)
〔3〕 R.B.Dannenberg and G.W.Ernst, "Formal Programing Verification Using Symbolic Execution", IEEE Trans. Softw. Eng., Vol.SE-8, No.1 (1982)
〔4〕 N.Wirth, "Program Development by Stepwise Refinement", Comm. ACM, Vol.14, No.4 (1971)
〔5〕 D.I.Good, R.L.London, and W.W. Bledsoe, "An Interactive Program Verification System", IEEE Trans. Softw. Eng., Vol.SE-1,No.2 (1975)
〔6〕 R.S.Boyer, B.Elspas and K.N.Levitt, "SELECT-A Formal System for Testing and Debuging Programs by Symbolic Execution", Int. Conf. on Reliable Softw. (1975)
〔7〕 L.Brownston,et al., "Programming Expert Systems in OPS5", Addison-Wesley (1985)
〔8〕 R.S.Boyer and J.S.Moore, "A Computational Logic",Academic Press (1979)
〔9〕 福島，永田，"プログラムの検証と再利用を考慮したプログラム作成支援システム"，情報処理学会 ソフトウェアー工学研究会資料,36-1 (1984)
〔10〕 岡田,二木，鳥居,"変換法によるプログラミング－階層的仕様記述に基づく一方法－",電子通信学会技術研究報告，EC81-76 (1982)
〔11〕 岡田,二木，鳥居,"階層的仕様記述による変換プログラミング－変換の型と正当性－",電子通信学会技術研究報告，AL82-42 (1982)
〔12〕 J.L.Bates and R.L.Constable, "Proofs as Programs", ACM TPLAS, Vol. 7, No.1 (1985)
〔13〕 雨宮,丸山,"関数型言語Validによる在庫管理システムの記述",情報処理，Vol.26, No.5 (1985)
〔14〕 B.H.Liskov and S.N.Zilles, "Programming with abstract data types", SIGPLAN Notices 9, 4 (1974)