論理型仕様からのストリーム処理プログラムの導出†

星野 寛†† 阿草清滋†† 大野 豊††

対象をストリーム型とした宣言的な論理型仕様をプログラム変換の手法を用いて手続き型プログラムに変換する手法を提案する．宣言的な仕様において問題の入出力データはリストに抽象化され，入出力リストとその間の関係として仕様は定義される．このように定義することにより，入出力データのタイプや構造およびその処理手順はその後の設計に委ねられることになり，純粋に問題を構成する対象間の関係で仕様定義が行える．入出力リスト間の関係はリスト構造を変換する簡単な述語の組合せで表現される．この述語はプランと呼ばれ，入力ストリームからある概念を切り出す方法を宣言的に定義している．プランの組合せで仕様を記述することにより，設計段階における小さな概念の再利用が可能となり，さらにプランを結合することにより新たな概念に対応するプランの合成ができる．プランの組合せで定義された仕様をプログラム変換により，入力データストリームを先頭から処理し出力リストを生成するような手順的な仕様に変換する．その後，入出力述語を付加し手続き型プログラムに変換する．一般に宣言的な仕様を実現するプログラムは複数考えられるが，本研究では対象をストリーム処理に限定することにより，宣言的な仕様から手続き型プログラムを導出するための変換戦略を明らかにした．ここでは本研究におけるストリーム処理プログラムの開発手法および例を示す．

1. まえがき

ソフトウェアの仕様を論理型言語で記述し，規則的な変換によって半自動的に手続き型プログラムを得る方法を提案する．

ソフトウェアの仕様化段階では入出力や処理の手順等の詳細なことには触れず，問題を解決するために処理対象や処理機能の関係を大局的にとらえる．一方，プログラミング段階ではそれらの関係を効率的に実現するための入出力や処理の手順が考慮される．効率化のために機能や処理対象がプログラム中に分散して存在しがちで，仕様記述との対応が取れずプログラムは理解しづらいものになることが多い．設計記述とプログラムの間のギャップを埋め，関係を明らかにすることができれば，設計仕様書からプログラムの自動合成や，プログラムの理解，保守への足掛りとなる．

ソフトウェアは小さな処理概念の組合せで構成され[1]，Waters や Soloway らはこの小さな処理概念をプランと呼び，プランを用いてプログラムが構成できることを示した[2)~4)]．しかし，Waters や Soloway らの研究ではどのようなプランを用意し，どのようにプランが結合されるかについては必ずしも明らかにされていない．本研究では，対象をストリーム処理に限定し，ストリームを入出力とするプランを用意することで，プランの結合を半自動化できることを明らかにする．入出力データはリストまたは単独の値で表現し，プランは入出力データ間の関係を表現する述語として用意した．このようなプランを用意しておくことによって，あるプランの出力データを次のプランの入力データに容易に結び付けることができ，プラン同士を容易に結合することができた．

2章では，本研究におけるプランの表現方法と，それらのプランを用いたソフトウェアの設計方法について述べる．本研究におけるプランはデータの入力順序に依存しない形で与えられている．本研究では，入力順序に依存しないソフトウェアの仕様を宣言的であると言い，入力順序が明らかになった仕様を手順的と呼ぶ．

次に，このようなプランの組合せ記述を設計仕様書とみなし，プログラム変換の手法[5)~10)]を用いて変換し，入出力の手順を明らかにすることにより，効率のよい手続き型プログラムを導出する．一般に宣言的な仕様を実現するプログラムは変換手法や変換戦略によって複数考えられ，どのような変換戦略をどの順序で適用したらよいのかは明らかになっていない．本研究では対象をストリーム処理に限定したことで，宣言的な仕様に対して手順的な解釈を与えるような変換戦略を用いて手続き型プログラムを導出できた．また，この変換過程において抽象的にリスト表現されていた入出力データが具体化される．

本研究における変換は次の3つの段階からなり，それぞれ3，4，および5章で説明する．

† Development Method of Stream Program from Logic Specification by HIROSHI HOSHINO, KIYOSHI AGUSA and YUTAKA OHNO (Department of Information Science, Faculty of Engineering, Kyoto University).

†† 京都大学工学部情報工学科

（1） 仕様化段階
（2） 仕様変換段階
（3） プログラム導出段階

仕様化段階ではプランを組み合わせて問題を定義すると共に，組み合わせられたプランを簡約化する．仕様変換段階では入出力データにアクセスするタイミングを明らかにする．次のプログラム導出段階では入出力手順の明らかになった仕様に入出力用述語を付加し，入力プリミティブをくくり出すような変換を行う．このような変換段階を経ると，最終的に入出力述語を含んだループ構造を持つ Prolog プログラムが得られ，簡単に手続き型プログラムを導出できる．

本研究では，ソフトウェアの仕様は各変換段階を通じて述語論理記述で表現される．述語論理に基づいた記述は宣言的に見えるということで仕様記述として使え，規則的な変換によって手順を明らかにできるからである．

以下の章で記述される Prolog 表現は DEC-10 Prolog の文法[11]に従った．

2. プランによる設計

この章では平均値を求める問題を例としてプラン[1)~4)]を説明し，小さな処理概念であるプランを結合していくことで簡単に全体の仕様が記述できることを示す．こうして記述された仕様は以下の章で述べる変換戦略によって手続き型プログラムに変換される．

次にプログラムは平均値を求めるC言語プログラムである（ただし，getdata() は入力データを1つ取り出す関数とする）．

```
sum = 0, count = 0;                              (2.1)
while((data = getdata()) != EOF)
        count++, sum += data;
average = sum / count;
```

このプログラムには次のまとまった処理が含まれており，各々固有の概念を表している．一般にこのような小さな概念をプランと呼ぶ．

```
sum = 0, sum += data      ……総和処理
count = 0, count++        ……計数処理
average = sum / count     ……除算処理
```

本研究ではこれらのプランを Prolog の述語で表現し，プランを結合することにより問題を仕様化する．ただし，この段階では入出力データ列はリストで表現し入出力手順は考えない．Prolog で表現することにより，この段階で仕様は実行可能であり，設計者が意図したプランかどうかの確認ができる．問題をストリーム処理に限定すると，その処理は，入力リストをある単位で分ける，分けられたリストになんらかの処理を適用する，適用された結果を連結し出力リストまたは単独の値を生成するという3つのオペレーションで定義できる．

ここでは，これらの3つの基本オペレーションを「分割」「作用」「連結」と呼ぶ．「分割」オペレーションは入力がリストの時には append という述語で表現され，単独の値の時には存在しない．「連結」オペレーションは処理された結果が部分リストなら，append で表現され，単独の値ならそれらの値を処理する演算子が対応する[13)]．

これらのオペレーションを用いて定義される述語および，それらの述語を結合して定義される述語をプランと呼ぶ．ただし，後の変換のために，「分割」から「作用」，「作用」から「連結」への遷移は決定的であり，これらのオペレーション間のバックトラックは起こらないものとする．append を，入力リストを自由な位置で「分割」する述語としてとらえることによりデータの入力順序に依存しない定義ができるため，このプランは宣言的である．

プランの入出力リストまたは単独の値は複数個許されるが，それらは1つのストリームとして解釈され，別々のストリームに分岐することはない．入力リストが2つ以上あると，「分割」オペレーションもそのリストの数だけ存在し，出力リストが2つ以上あると，「連結」オペレーションも出力リストの数だけ存在することになる．

設計者はあらかじめ用意されているプランから問題を解くために適当なものを選び，それらのプランを組み合わせて問題の仕様記述を行う．この時，プラン同士のインタフェース条件，すなわち入出力リストの数とタイプ（リストか単独の値か）がチェックされる．

例えば，平均値を求める問題のプランによる仕様は次のようになる．

平均プラン

```
average(InputVals, Average) :-                   (2.2)
        sum(InputVals, Sum),
        count(InputVals, Count),
        divide(Sum, Count, Average).
```

総和プラン

```
sum([], 0). sum([OneVal], OneVal).               (2.3)
sum(InputVals, Sum) :-
```

「分割」 append(SHead, STail, InputVals),
「作用」 sum(SHead, HeadSum),
sum(STail, TailSum),
「連結」 Sum is HeadSum + TailSum.

計数プラン

count([], 0). count([OneVal], 1). (2.4)
count(InputVals, Count) :-
「分割」 append(CHead, CTail, InputVals),
「作用」 count(CHead, HeadCount),
count(CTail, TailCount),
「連結」 Count is HeadCount + TailCount.

除算プラン

divide(Sum, Count, Average) :- (2.5)
「連結」 Average is Sum / Count.

計数プラン (2.4) では「分割」「作用」「連結」というオペレーションは append, count, '+' という述語で表現され，入力ストリーム InputVals と出力 Count の関係を定義している．また，(2.2)では平均プランは総和，計数および除算プランを用いて定義されることを示している．このプランに対して4章，5章で述べるような手続き的な解釈を与える変換戦略を適用することによって，(2.1)のようなプログラムが得られる．この様子を図1に示す．

3. 仕様化段階

仕様化段階では2章で述べたプランを組み合わせて問題の仕様化を行い，さらに結合されたプランを一つのプランに簡約化する．この簡約化を行うことにより，より簡潔なプランを導出でき，入出力データ間の関係がより直接的になるため，次の段階における変換が容易になる．また，簡約化されたプランが新たな処理概念を表しているなら，簡約化される前のプランと共にプランデータベースに格納し，再利用される．

プランの結合には並列結合と直列結合がある．この章では，3つの基本オペレーションで宣言的に定義されたプランが並列または直列に結合されている場合の簡約化について述べる．また，新たに合成されたプラン自身3つの基本オペレーションで定義され，他のプランと結合簡約可能なことを示す．

3.1 並列結合されたプランの簡約化

2つのプランに対して同じ入力ストリームが与えられるとき，この2つのプランは並列に結合されているという．

例えば，2章で述べた平均値を求めるプランでは総和プラン (sum) と計数プラン (count) の入力が共にInputVals であり，総和プランと計数プランは並列に結合されている．

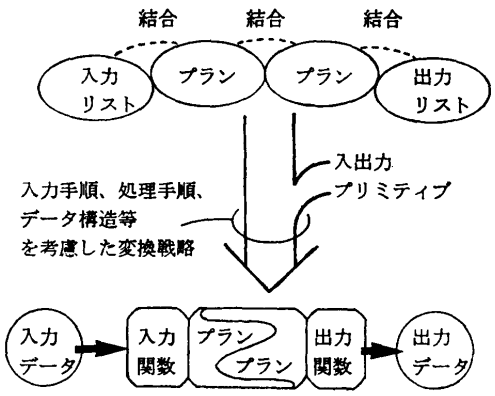


図1 プランによる設計
Fig. 1 Software design with "plan".

並列に結合されている2つのプランは共に，同一入力ストリームに対する「分割」オペレーションを持ち，2つの出力ストリームは1つに合流している．どちらのプランにおいても「作用」から「分割」オペレーションへのバックトラックが無く，「分割」オペレーションを表現する2つの述語が単一化可能ならば，「分割」オペレーションを1つにまとめ，2つのプランを1つに簡約化することができる (図2(a))．この簡約化は，入力ストリームに対する2つの繰り返し処理をマージすることに対応する．

例えば，(2.3)と(2.4)のプランを並列に結合した次の sum_count というプランを簡約化することを考える．

sum_count(InputVals, Sum, Count) :- (3.1)
sum(InputVals, Sum),
count(InputVals, Count),

sum, count を (2.3)，(2.4) の定義を用いて展開した後，(2.3)，(2.4)において下線の引かれた「分割」オペレーションである append という述語を単一化する．その結果 SHead と CHead, STail と CTail という変数を等価と見ることができる．次に sum_count で畳み込みを行うと，次のような新しいプランに簡約化できる．この場合，Sum と Count を求めるループをマージしたことになる．また sum_count 自身3つの基本オペレーションで定義されており，他のプランと簡約化可能である．

sum_count([ ], 0, 0). (3.2)
sum_count([Oneval], Oneval, 1).

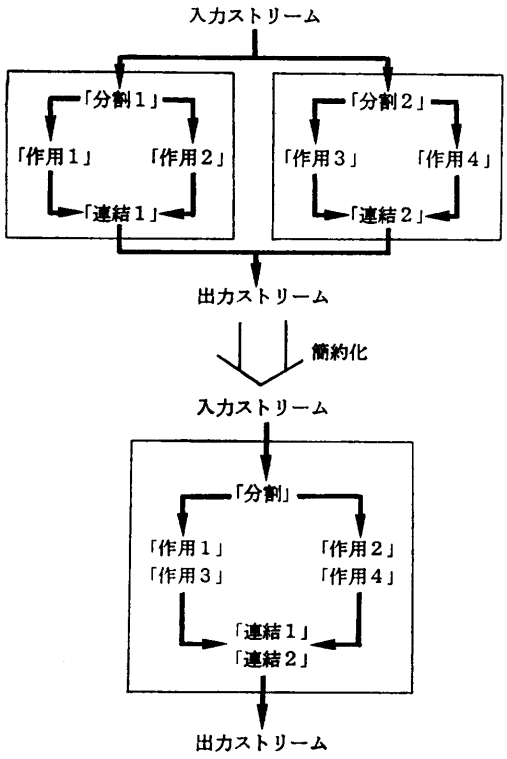


図2(a) 並列結合されたプランの簡約化
Fig. 2 (a) Reduction of parallel-connected "plans".

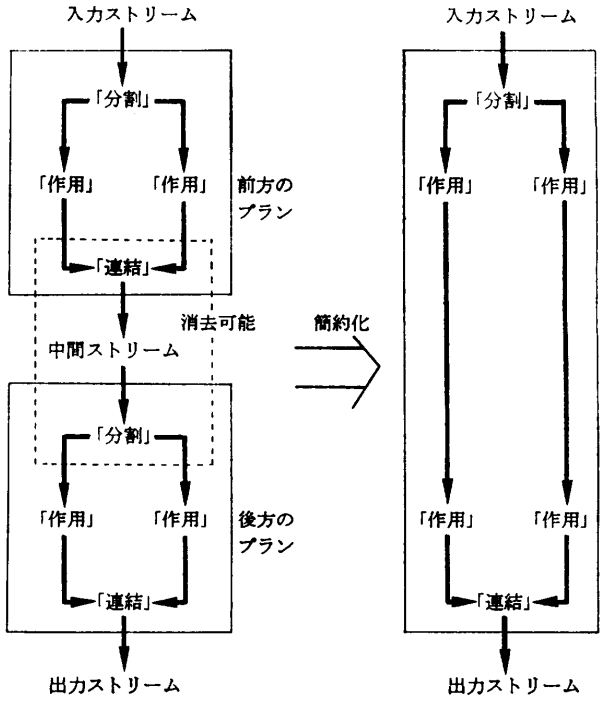


図2(b) 直列結合されたプランの簡約化
Fig. 2 (b) Reduction of sequential-connected "plans".

```
sum_count(InputVals, Sum, Count) :-
「分割」   append(Head, Tail, InputVals),
「作用」   sum_count(Head, HSum, HCount),
           sum_count(Tail, TSum, TCount),
「連結」   Sum is HSum + TSum,
           Count is HCount + TCount.
```

3.2 直列結合されたプランの簡約化

2つのプランにおいて一方のプランの出力ストリームがもう一方の入力ストリームになっているとき，このストリームを中間ストリームと呼び，2つのプランは直列に結合されているという．

直列に結合されているプランにおいて，前方のプランの「連結」オペレーションと後方のプランの「分割」オペレーションが連続する（**図2(b)**）．図2(b)において入力ストリームと出力ストリームの間に構造の不一致[12]が無く，連続する「連結」「分割」オペレーションを表現する述語が単一化可能ならば，この「連結」「分割」オペレーションを消去し，中間ストリームを生成せず，直接入力ストリームと出力ストリームを関係付けることができる．構造の不一致がある場合にはこの簡約化はできないが，4章で述べる変換を各プランに施した後に統合化できる．

例として，簡単なワードカウントを考える．ワードカウントは単語切出しプランと計数プランを直列に結合することによって構成でき，下のように記述できる[14]．ただし，ここでは変換を簡単にするために単語は空白文字のみで区切られるものとした．また，space Character(C) は C が空白文字がどうか調べるための述語である．

ワードカウントプラン

```
wordcount(List, Count) :-                    (3.3)
        getword(List, Words),
        count(Words, Count).
```

単語切出しプラン

```
getword([ ], [ ]).                           (3.4)
getword(List, Words) :-
「分割」   append(WHead, [C|WTail], List),
           spaceCharacter(C),
「作用」   getword(WHead, HWords),
           getword(WTail, TWords),
「連結」   append(HWords, TWords, Words).
getword(List, [List]).
```

この場合，単語切出しプランの出力が計数プランの入力となっている．ここで，(3.3)の getword と

count を(3.4)，(2.4)の定義で展開すると次のようになる．

```
wordcount([ ], 0).                                    (3.5)
wordcount(List, Count) :-
「分割」  append(WHead, [C|WTail], List),
          spaceCharacter(C),
「作用」  getword(WHead, HWords),
          getword(WTail, TWords),
「連結」  append(HWords, TWords, Words),
「分割」  append(CHead, CTail, Words),
「作用」  count(CHead, HeadCount),
          count(CTail, TailCount),
「連結」  Count is HeadCount + TailCount.
wordcount(List, 1).
```

下線のついている「連結」の append は getword によって文字列から単語が認識される順序を表しており，「分割」の append は単語の数を計数する順序を表している．この時，2つの append は単一化でき，引数 HWords と CHead, TWords と CTail を等価と見ることができる[10]．その結果，「連結」と「分割」は消去でき，次のような述語が得られる．

```
wordcount([ ], 0).                                    (3.6)
wordcount(List, Count) :-
「分割」  append(WHead, [C|WTail], List),
          spaceCharacter(C),
「作用」  getword(WHead, HWords),
          getword(WTail, TWords),
「作用」  count(HWords, HeadCount),
          count(TWords, TailCount),
「連結」  Count is HeadCount + TailCount.
wordcount(List, 1).
```

ここで getword, count という述語の並び替えを行い，この getword, count というペアを wordcount で畳み込むと，次のような入力文字列と語数を直接関係付けるプランを得ることができる．また，語数をカウントするためには入力文字列を保存しておく必要がないことも，(3.7)のプランの簡単なデータフロー解析をすることによって発見できる．

```
wordcount([ ], 0).                                    (3.7)
wordcount(List, Count) :-
「分割」  append(Head, [C|Tail], List),
          spaceCharacter(C),
「作用」  wordcount(Head, HeadCount),
          wordcount(Tail, TailCount),
「連結」  Count is HeadCount + TailCount.
wordcount(List, 1).
```

(3.2)における「分割」オペレーションを表現する述語の単一化や，(3.5)における連続する「連結」「分割」オペレーションを表現する述語の消去ができたのは，count の「分割」オペレーションがリストのどこで分割してもよいような述語 (append) として定義されていたために sum の「分割」オペレーションや getword の「連結」オペレーションと同期を取ることができたからである．

すなわち，(2.3)，(2.4)，(3.4)のように3つの基本オペレーションで定義されているプランは入力リストに対する処理手順を append という述語に隠ぺいしているために他のプランとの間で append の単一化，消去を行うことにより，簡約化が行える．

4. 仕様変換段階

前段階で，append を使って入力リストを分割するという宣言的な仕様記述が得られた．仕様変換段階では，この宣言的な仕様記述を，入出力のタイミングを明らかにするために，入力リストの先頭から順番に処理する手順的な仕様に変換すると共に，いつ出力リストの各要素の値が確定されるかを明らかにするような変換が行われる．このように変換することにより，手続き型プログラムとのギャップを埋めることができ，次の手続き型プログラム導出段階において入出力プリミティブを挿入する位置を明らかにすることができる．

入力リストの先頭から順に処理するような仕様は，「分割」オペレーションを手順的に再定義することにより導出できる．出力リストの各要素の値が確定する時期を明らかにするためには「連結」オペレーションに注目し，各要素が確定するまで中間値を保存するためのアキュムレーティングパラメータを持つように述語を再定義する．ただし，この段階では「作用」から「分割」，「連結」から「作用」へのバックトラックは起こらないことを仮定しなければならない．

4.1 「分割」オペレーションの再定義

プランにおいて「分割」から「作用」へのバックトラックは起こらないものと仮定すると，「分割」の append を手順的に解釈すれば，手順的な仕様が得られる．append の定義は(4.1)のように手順的であり，本研究におけるプランでは入力リストへのアクセス手順はすべてこの append で決まるからである．

```
append([ ], L, L).                                    (4.1)
append([H|T], L, [H|R]) :- append(T, L, R).
```

例えば，(3.1) の sum_count において「分割」の append を展開すると，次の定義が得られる．

```
sum_count([ ], 0, 0).                                 (4.2)
sum_count([H|Tail], Sum, Count) :-
        sum_count(Tail, TailSum, TailCount),
        Sum is H + TailSum,
        Count is 1 + TailCount.
```

また，「分割」オペレーションが append と分割条件を表す他の述語で定義されているような場合は，この append と分割条件のペアを入力要素を足し込んでいくためのアキュムレーティングパラメータを持つ述語に再定義する．このように再定義することによって入力リストの先頭から順番に，分割条件が成立するかどうかを調べる手順的な仕様が得られる．

例えば，(3.7)では「分割」オペレーションはappendと分割条件 spaceCharacter(C) で定義されているが，このペアを次のようにアキュムレーティングパラメータを持つ述語に再定義する．

```
appendl(H, [C|T], [C|T], H) :-                        (4.3)
        spaceCharacter(C).
appendl(H, T, [N|L], Temp) :-
        append(Temp, [N], Tmp),
        appendl(H, T, L, Tmp).
```

すると wordcount は次のようになる．

```
wordcount([ ], 0, [ ]).                               (4.4)
wordcount(List, Count, Temp) :-
        appendl(H, [C|T], List, Temp),
        wordcount(H, HeadCount, [ ]),
        wordcount(T, TailCount, [ ]),
        Count is HeadCount + TailCount.
wordcount(List, 1, [ ]).
```

さらに appendl を展開すると次のような述語が得られ，入力リストの先頭から順にスキャンしながら単語を認識し，語数をカウントしていることがわかる．

```
wordcount([ ], 0, [ ]).                               (4.5)
wordcount([ ], 1, Temp).
wordcount([C|T], Count, [ ]) :-
        spaceCharacter(C),
        wordcount(T, Count, [ ]).
wordcount([C|T], Count, Temp) :-
        spaceCharacter(C),
        wordcount(T, TailCount, [ ]),
        Count is 1 + TailCount.
wordcount([N|L], Count, Temp) :-
        append(Temp, [N], Tmp),
        wordcount(L, Count, Tmp).
```

また，3.2 節で発見したように，語数をカウントするために入力文字列を保存しておく必要はないので，入力文字列を保存するための Temp という変数は単に単語を構成する文字を読んだかどうかのフラグであればよい．Temp をフラグと解釈すると次のような定義が得られる．

```
wordcount([ ], 0, 0).                                 (4.6)
wordcount([ ], 1, 1).
wordcount([C|T], Count, 0) :-
        spaceCharacter(C),
        wordcount(T, Count, 0).
wordcount([C|T], Count, 1) :-
        spaceCharacter(C),
        wordcount(T, TailCount, 0),
        Count is 1 + TailCount.
wordcount([N|L], Count, Flag) :-
        wordcount(L, Count, 1).
```

4.2 「連結」に注目した述語の再定義

出力リストの各要素の値が確定する時期を明らかにするためには，出力リストの各要素の値が決定するまで中間値を保存するアキュムレーティングパラメータを用意すればよい．例えば，(4.6)において Count が出力であるが，この値が最終的に決定するまでCTempという変数に中間値を足し込むように wordcount を再定義すると次のようになる．

```
wordcount([ ], Count, 0, Count).                      (4.7)
wordcount([ ], Count, 1, CTemp) :-
        Count is CTemp + 1.
wordcount([C|T], Count, 0, CTemp) :-
        spaceCharacter(C),
        wordcount(T, Count, 0, CTemp).
wordcount([C|T], Count, 1, CTemp) :-
        spaceCharacter(C),
        CTmp is 1 + CTemp,
        wordcount(T, Count, 0, CTmp).
wordcount([H|L], Count, Flag, CTemp) :-
        wordcount(L, Count, 1, CTemp).
```

5. プログラム導出段階

プログラム導出段階においては，仕様変換段階で得

られた仕様記述に入出力用述語を付加し，さらに変換を行うことによって入出力手順が明らかになったプログラム仕様を得る．ここで，入力用述語とは入力データを読み込むプリミティブ述語を使って入力データを入力リストに組み上げる述語であり，出力用述語とは出力データを出力するプリミティブ述語を使って出力リストの先頭から順に出力するための述語である．仕様変換段階で得られる仕様記述は入出力のタイミングが明らかになっているため，実際に入出力処理を行う述語と結合でき，自動的な変換が可能となっている．

ここで問題となるのは，入出力プリミティブに対してはバックトラックが許されないために，Prolog の ':-' を等号と考え展開／畳み込みを行うという変換が適用できないことである．本研究では，入力プリミティブをくくり出し，入力プリミティブとそれ以外の部分に分けて変換することでこの問題を解決した．これは入力用述語の中に入力プリミティブが局所化されていることによって可能となっている．

入出力手順が明らかになったプログラム仕様では，入力データを1つずつ取り込み，処理をし，出力データを得るというループ構造を発見できる．ここで各述語のヘッダ部および，「分割」オペレーションにおける分割条件を表す述語を手続き型言語の条件文に対応させることによって，最終的にストリーム処理を行うループ構造を持ったプログラムが得られる．

以下の部分では wordcount を例にして入出力処理まで含めて手順的なプログラム仕様が得られることを示す．

ここで，ワードカウントプログラムへの入力は文字列であり，その文字列はファイルに格納されているものとする．また，出力は数値である．するとファイルから文字を読み込むための get 0 と数値を書き出すための write という Prolog の入出力プリミティブを使って入出力用述語を次のように定義できる．

入力用述語

```
readfile(InputFile, List) :-                    (5.1)
        see(InputFile),
        getlist(List), seen.
getlist(R) :- get0(X), conschar(X, R).
conschar(eof, [ ]).
conschar(X, [X|R]) :- getlist(R).
```

出力用述語

```
write(Count).                                   (5.2)
```

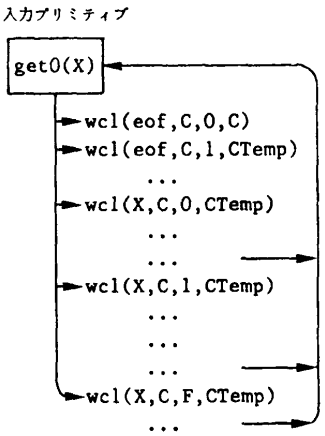
これらの入出力用述語を(4.7)の wordcount の前後に付加するとワードカウントは次のように定義される．

```
wc(InputFile) :-                                (5.3)
        readfile(InputFile, List),
        wordcount(List, Count, 0, 0),
        write(Count).
```

ファイルのオープン，クローズを行う see, seen を述語の最初と最後に移動し，入力プリミティブ get 0 をくくり出す形で変換を行うと，**図 3**(a)のような Prolog プログラムが得られる．ここで wcl が入力プリミティブを含まない述語になっている．このプログ

```
wc(InputFile) :-
    see(InputFile),
    wordcount(Count,0,0),
    write(Count),seen.
wordcount(C,F,CTemp) :-
    get0(X),
    wcl(X,C,F,CTemp).
wcl(eof,C,0,C).
wcl(eof,C,1,CTemp) :-
    C is CTemp + 1.
wcl(X,C,0,CTemp) :-
    spaceCharacter(X),
    wordcount(C,0,CTemp).
wcl(X,C,1,CTemp) :-
    spaceCharacter(X),
    CTmp is 1 + CTemp.
    wordcount(C,0,CTmp).
wcl(X,C,F,CTemp) :-
    wordcount(C,1,CTemp).
```

(a) Prolog 記述



(b) 制御構造図

```
wordcount()
{
int     c,count,flag;
        for(count = flag = 0;;) {
                c = getc(ifp);
                if (c == EOF && flag == 0)
                        return(count);
                else if (c == EOF && flag == 1)
                        return(count + 1);
                else if (spaceCharacter(c) && flag==0)
                        ;
                else if (spaceCharacter(c) && flag==1)
                        count++, flag = 0;
                else flag = 1;
        }
}
```

(c) C言語による記述

図 3 ワードカウントプログラムとその制御構造
Fig. 3 Word count program and its control flow.

ラムはテイルリカーシブな構造を持つが，これをループ構造としてみると図3(b)のような制御構造が得られる．さらに spaceCharacter(C) と述語のヘッドを条件部と考え，Count と CTemp を一つの変数 count で表現すると，簡単に図3(c)の手続き型プログラムが得られる．

6. むすび

本論文ではソフトウェアは小さな概念の組合せで構成されているという立場から，仕様化段階からプログラム導出段階における設計者およびプログラマの作業をプランの結合とプログラム変換で説明付けた．特に問題領域をストリーム処理に限定し，この変換作業が機械的に行えることを示した．

本研究は，プランをソフトウェア部品と考えるとソフトウェアの部品化による再利用の研究と似ているが，プランは最終的なプログラムの中に分散して存在するところが異なる．すなわち，従来のソフトウェア部品では小さすぎて扱えなかった処理概念やプログラム中に散在するプログラミングのノウハウをプランという単位で再利用したことになる．

現在，この作業を支援するためのシステムを開発中であるが，プランの検索・組合せおよび入出力用述語の検索・定義は設計者が行い，それ以外の変換作業を自動化することを目標としている．また，どのようなプランを用意しどのように検索すればよいか等問題点は残されている．

文献15)に本研究の方法を電報解析問題に適用した際の報告を行った．

謝辞 本研究を進めるにあたって，ソフトウェアの設計作業とプログラム変換の関係についてディスカッションし，ヒントを与えてくださった京都大学情報工学科助手鯵坂恒夫氏に感謝いたします．また，日頃から本研究テーマについて様々な批判，議論をしてくださった京都大学大野研究室の方々に感謝いたします．

参考文献

1) Waters, R. C.: A Method for Analyzing Loop Programs, *IEEE Trans. Softw. Eng.*, Vol. SE-5, pp. 237-247 (May 1979).
2) Waters, R.C.: *The Programmer's Apprentice: Knowledge Based Program Editing Interactive Programming Environments*, pp. 464-486, McGraw-Hill Book Company, New York (1984).
3) Johnson, W. L. and Soloway, E.: PROUST: Knowledge-Based Program Understanding, *7th ICSE*, Mar. 1984, Orlando, Florida.
4) 上原邦昭，藤井邦和，豊田順一：自然言語による仕様からの自動プログラム合成，コンピュータソフトウェア，Vol. 3, No. 4, pp. 55-64 (Oct. 1986).
5) Darlington, J.: The Structured Description of Algorithm Derivation, *Algorithmic Languages*, de Bakker/van Vliet (eds.) IFIP, pp. 221-250, North-Holland Publishing Company, New York (1981).
6) Clark, K. and Sickel, S.: Predicate Logic: A Calculus for Deriving Programs, *5 th IJCAI*, pp. 419-420 (1977).
7) Hogger, C. J.: Derivation of Logic Programs, *JACM*, Vol. 28, No. 2, pp. 372-392 (April 1981).
8) 佐藤泰介，玉木久夫：論理プログラムの等価変換とプログラム合成への応用，情報処理，Vol. 26, No. 11, pp. 1423-1431 (1985).
9) 玉木久夫，佐藤泰介：Prolog における Append プログラミング，第28回情報処理学会全国大会論文集，pp. 391-392 (1984).
10) 中川裕志：木構造を扱うアルゴリズムの変換による合成，情報処理学会論文誌，Vol. 26, No. 5, pp. 870-876 (1985).
11) Pereira, L. M., Pereira, F. and Warren, D.: *User's Guide to DEC System-10 Prolog*, Laboratorio Nacional de Engenharia Civil, Lisbon, Portugal (1979).
12) Jackson, M. A.: *Principles of Program Design*, Academic Press, London (1975).
13) 江指正洋，星野 寛，阿草清滋，大野 豊：図を用いた論理型プログラムの設計，第34回情報処理学会全国大会論文集，pp. 919-920 (1987).
14) 星野 寛，江指正洋，阿草清滋，大野 豊：論理型言語を用いたソフトウェア開発，日本ソフトウェア科学会，SW-87-1-3 (1987).
15) 星野 寛，阿草清滋，大野 豊：論理型言語に基づいた形式的なソフトウェア開発方法，日本ソフトウェア科学会第3回大会論文集，pp. 237-240 (1986).

(昭和62年8月3日受付)
(昭和62年11月11日採録)

星野 寛（正会員）

1958年生．1981年京都大学工学部情報工学科卒業．1984年同大学院修士課程修了．1987年同大学院博士課程退学．現在，京都大学工学部情報工学科研究生．ソフトウェア工学，論理型プログラミング，ソフトウェア開発環境，オペレーティングシステム等の研究に従事．1987年情報処理学会第34回全国大会学術奨励賞受賞．日本ソフトウェア科学会会員．

阿草 清滋（正会員）

1947年生．1970年京都大学工学部電気第二学科卒業．1972年同大学院工学研究科電気工学第二専攻修士課程修了．1974年より京都大学工学部情報工学科に勤務．1986年から87年までカリフォルニア大学客員研究員．現在，助教授．工学博士．ソフトウェア工学に関する研究に従事．とくに要求分析，ソフトウェア仕様化技法，ソフトウェア部品化技法，ソフトウェア開発モデルなどに興味をもつ．また，マンマシンシステム，分散処理システム等に関する研究も行っている．1985年度情報処理学会論文賞．電子情報通信学会，日本ソフトウェア科学会会員．

大野 豊（正会員）

1924年生．1946年東京大学工学部機械工学科卒業．同年より国鉄鉄道技術研究所に勤務，座席予約システム，新幹線運転管理システムなどの研究・開発に従事．1972年より京都大学工学部情報工学科教授，情報システム工学講座担任．工学博士．京都大学情報処理教育センター長を兼任．現在，ソフトウェア工学，システム性能評価，分散データベース，コンピュータグラフィックスなどの研究を行っている．1960年電気学会進歩賞，1968年電子通信学会業績賞，1971年紫綬褒章，1975年情報化個人表彰，1985年度情報処理学会論文賞．計測自動制御学会，日本機械学会等の会員．本学会理事，副会長，計測自動制御学会理事，第3回日米コンピュータ会議議長，第6回ソフトウェア工学国際会議議長，第12回巨大データベース国際会議議長等を歴任．1983年より日本ソフトウェア科学会理事長．1985年よりシグマ・システム開発委員会委員長．著書「オンラインリアルタイムシステムの計測」ほか．