ソフトウェア基礎論　38－8
プログラミング言語　28－8
（１９９１．３．８）

関数型言語を用いたパタン駆動に基づくシステム実現手法
－　並列関数型言語処理系の記述と実現　－

中山　仁*　　中村　浩**　　荒木　啓二郎**

*九州工業大学 情報科学センター
**九州大学 工学部 情報工学科

関数型言語を用いたシステム実現手法の１つとして，パタンとパタンマッチングの機構に基づくものを提案する．この方式ではシステムが取り扱う対象をパタンの集合とみなし，システム自体を各パタンに対する規則の集合として，パタンマッチングによって実行を制御する．本稿ではまず，この手法を適用して簡単な関数型言語処理系の実現を行い，その有効性を検討する．さらに，パタンとして取り扱えるようなデータ構造を処理の対象としない，一般のシステムに対して，システムで発生する事象をパタンとみなしてパタン駆動手法を適用する方法を述べ，例として哲学者の食事問題を記述する．なお記述言語としては，関数型言語 Miranda を用いた．

Pattern-Driven Approach to System Development Using Functional Languages
- Implementation of the Parallel Functional Language System -

Hitoshi NAKAYAMA*, Hiroshi NAKAMURA** and Keijiro ARAKI**

*Information Science Center, Kyushu Institute of Technology
**Department of Computer Science and Communication Engineering, Kyushu University

We propose the system development approach using functional languages based on patterns and pattern-matching. In this pattern-driven approach, objects processed by systems are considered as set of patterns, and systems are described as set of rules corresponding to each patterns, and the system behavior is determined by the pattern-matching mechanism. We apply this approach to development of a simple functional language system and examine its power. And then, we explain the method to apply the pattern-driven approach to the systems that do not treat objects that can be considered as patterns, and specify the dining philosopher problem as an example.

1　はじめに

関数型プログラムは，問題の解法を高いレベルで簡潔に記述できる，数学的な取り扱いが容易であり，検証や変換が比較的容易である，などの多くの特徴を持つ．これらはいづれも大規模なシステムの開発における生産性や信頼性の向上につながるものであり，その意味で，関数型プログラムは大規模システムの記述に適していると言える．

しかしこれまで関数型プログラムをそうしたシステムの開発に用いた例は少なく，開発手法や方法論についても十分確立されているとは言えない．関数型プログラムによる開発の有効性と問題点を明らかにするために，実用規模の例題による事例研究が必要である．

一方，関数型プログラムには，内在する並列性を抽出し，利用することが比較的簡単であるという特徴もあるため，多くの並列実行方式が提案，研究されている．筆者らも比較的低い並列度のマルチプロセサあるいは分散処理環境上で効率的に動作することを目標とした，結合子グラフ簡約に基づく並列実行方式を提案し，研究を行っている．この実行方式を具体化し，検討や評価を行うための実験環境として簡単な関数型言語処理系を作成する必要が生じてきた．

この実験用処理系の要求としては，

・実現が容易であること
・システムの（仕様）変更が容易であること
・効率や性能は度外視できること

などを挙げることができる．これらを満足するため，システム記述言語として関数型言語 Miranda を用いることとし，さらに実験処理系の実現と同時に，関数型プログラムによるシステム記述の事例研究を行うことにした．

言語処理系にとって，ソースプログラムを構文パタンの組み合わせとして考えることができることから，本システムでは，システムを構文パタンとそれに対応する動作の形で記述しておき，プログラム中のパタンにマッチしたものから次々に実行していく，パタン駆動の手法[1]を試みた．

本稿では，このパタン駆動アプローチによる言語処理系の実現法について述べ，さらにより一般的な問題に対するパタン駆動の適用手法について説明する．

以下，２節では今回実現する関数型言語処理系の概要を説明し，３節で具体的な実現手法について述べる．４節ではより一般的な問題に対するパタン駆動手法の適用方法について，簡単な例題を用いて説明し，さらに本処理系の並列実行部の記述への応用について述べる．５節では今後の課題等を述べる．

2　pfp 処理系の概要

2.1　関数型言語 pfp と Miranda

今回処理系を作成する pfp およびその記述に用いる Miranda は，ともに SASL や KRC などの言語の流れをくむ関数型言語である．両者は構文的にはかなり異なっているが，機能的には pfp は Miranda に対してほぼサブセットになっている．共通の特徴としては以下のようなものがある．

・高階関数
・強い型付け，多相型
・遅延評価

pfp プログラムは関数定義と，評価すべき式とで構成される．式としては各種データ演算，関数適用，if-then-else，および lambda 抽象がある．基本的なデータ型としては，整数，文字，論理値の３つがあり，さらにリスト型，組（tuple）型，関数型が用意されている．演算は基本型の要素に対する演算（四則演算，比較など）および，基本リスト操作がある．

並列実行系の実験という目的のため，実験用言語は言語仕様が簡単で実現が容易である一方で，ある程度の規模の問題に対応できる記述能力が必要である．上記の pfp の仕様は，このような点からみて十分なものであると判断した．

一方 Miranda は pfp に加えてさらに多くの機能を持つが，今回は，

・代数データ型，抽象データ型，型変数
・パタンマッチング

などを多用した．またプログラミング環境として，基本的な対話環境やオペレーティングシステムとのインターフェースの機能も備えている．

2.2　システムの構成

図２.１に本処理系の全体構成を示す．システムは，原始プログラムを抽象構文表現に変換するパーザ部および，その抽象構文表現を入力とする型検査部，実行部，そして並列結合子グラフ生成部からなる．

パーザ部は，pfp 原始プログラムを入力し，これに対して字句解析と構文解析を行って，抽象構文表現されたプログラムを出力する．

pfp の抽象構文は Miranda のデータ構造（代数データ型）を用いて図２.２のように表現する．この抽象構文の定義はもとの pfp 構文の定義とよく対応しており，導出は容易である．抽象構文表現の例として，図２.３(a) の pfp プログラムを抽象構文表現に変換したものを図２.３(b)に示す．

型検査部，実行部，並列結合子グラフ生成部はすべて，この抽象構文表現されたプログラムを入力として動作する．なお，実行部は逐次実行を行うインタプリタであり，並列実行方式の実験という目的からすると本質的には必要でない部分であるが，並列実行結果の検定等を行うために用意することにした．

最後に，並列結合子グラフ生成部から出力される結合子グラフを簡約し，プログラムの実行を行う並列簡約系がある．最終的にこの部分は実際の並列／分散ハードウェア上

に，Cなどの言語を用いて実現する予定であるが，その前段階として，基本的な動作の確認と仕様の確定を行うため，Miranda を用いた実現を行う予定である．

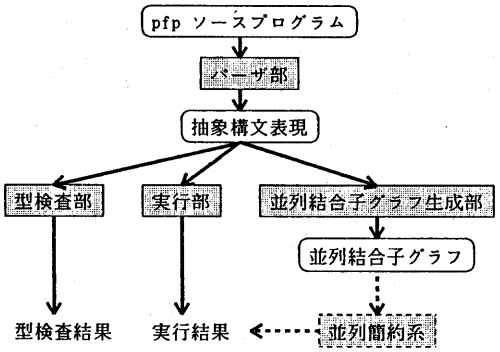

図2．1　システム構成

2．3　並列簡約方式の概要

pfp 処理系で実現しようとしている並列処理方式[6]は，並列制御結合子（PRCC）と呼ばれる一種の拡張された結合子を用いたもので，並列度の比較的低いハードウェア環境での実行効率の改善をめざすものである．

PRCC[6]は，いわゆる Turner の結合子の簡約規則を拡張して部分結合子式の簡約の起動を制御する情報を持たせたものである．たとえば，従来の S 結合子の簡約規則は，

```
S x y z → x z (y z)
```

であるが，PRCC ではこれが，

```
S0 x y z → (x z)* (y z)
S1 x y z → (x z*)* (y z*)
S2 x y z → (x z)* (y z)*
S3 x y z → (x z*)* (y z*)*
```

のようになる．ここで＊印が付いているのが次のステップで簡約を開始する（してもよい）部分式である．

PRCC の生成アルゴリズムの方針は，正規戦略を用いて簡約を行った場合に確実に簡約候補（リデックス）になる部分式を抽出し，それができるだけ早い時点で簡約開始されるようにする，ということである．こうしたリデックスとしては，正規簡約を行った場合に，将来必ず式の最左最外となる「位置」にある式，およびそのような式にとって必須な引数がある．前者はプログラムの構造の解析，後者は

```
||***** pfp_abstract_syntax *****
string       == [char]
int          == num
p_bool       == bool

pfp_program  ::= Abs_pfp_program constructors

constructors ::= Abs_empty_program
                 |Abs_func_def binding constructors
                 |Abs_expr expr constructors

binding      ::= Abs_binding ident expr

expr         ::= Abs_constant int
                 |Abs_expr_id ident
                 |Abs_negation p_bool
                 |Hd expr
                 |Tl expr
                 |Null expr
                 |Atom expr
                 |Abs_func_appli expr expr
                 |Abs_minus expr
                 |Abs_mult expr expr
                 |Abs_div expr expr
                 |Abs_mod expr expr
                 |Abs_conj expr expr
                 |Abs_add expr expr
                 |Abs_sub expr expr
                 |Abs_disj expr expr
                 |Abs_list_con expr expr
                 |Abs_list_app expr expr
                 |Abs_atomic_eq expr expr
                 |Abs_atomic_ineq expr expr
                 |Abs_less expr expr
                 |Abs_greater expr expr
                 |Abs_lesseq expr expr
                 |Abs_greatereq expr expr
                 |Abs_if expr expr expr
                 |Lambda ident expr
                 |Expr_list expr_list
                 |Expr_ntuple expr_ntuple

ident        ::= Abs_id string

expr_list    ::= Abs_empty_expr_list
                 |Abs_expr_list expr expr_list

expr_ntuple  ::= Abs_couple_expr expr expr
                 |Abs_expr_ntuple expr_ntuple expr
```

図2．2　pfp の抽象構文定義

```
def sort x = if null x then [] else insert (hd x) (sort (tl x))
```

（a）pfp プログラムの例（部分）

```
(Abs_binding (Abs_id "sort")
    (Lambda (Abs_id "x")
        (Abs_if (Null (Abs_expr_id (Abs_id "x")))
            Abs_empty_expr_list
            (Abs_func_appli (Abs_func_appli (Abs_expr_id (Abs_id "insert")) (Hd (Abs_expr_id (Abs_id "x"))))
                            (Abs_func_appli (Abs_expr_id (Abs_id "sort")) (Tl (Abs_expr_id (Abs_id "x"))))
        ))))
```

（b）(a) を抽象構文表現したもの．

図2．3　抽象構文表現の例

ストリクト性解析[7]を用いて抽出する．

さて，PRCC のような Turner 型の結合子の簡約は，簡約処理が非常に単純なのに対して，簡約ステップ数が大きい．PRCC のような並列簡約の場合，これは小さなプロセスが大量に発生することに相当する．そのため，比較的並列度が低く通信コストの高い，汎用プロセサによるマルチプロセサなどで PRCC 簡約を実現しようとすると，プロセス管理やプロセス間通信などのオーバヘッドによって十分な効率が得られないおそれがある．

そこで今回の pfp の実現方式では，PRCC を部分的に使用することにした．すなわち，プログラムをいくつかの並列処理単位（関数，正確にはスーパーコンビネータ）に分割し，それらへのデータ（引数）の分配と起動制御のみを PRCC によって行う．この方式では，並列処理単位への分割をどのようにして行うかが問題となる．各プロセス間の負荷のバランスなどを厳密に考慮するならば，分割のための手続きを作成することは非常に難しい問題（静的な負荷分散の問題）である．今回はごく簡単な規則を設定して，粗い近似をとることにした．

この並列簡約系を実装するハードウェアアーキテクチャなどについては，現在のところ特に規定していない．考えうるいくつかのモデルについて実現を行い，評価を行うことも今後の実験の目的の１つである．

3　パタン駆動手法による言語処理系の実現

pfp 処理系は図２．１のように複数のサブシステムから成り立っているが，それらのうち抽象構文表現されたプログラム（抽象プログラム）を入力とするサブシステム（型検査部，実行部，並列結合子グラフ生成部．グラフ生成部はさらにストリクト性解析部，静的負荷解析部を含む）は全て抽象構文定義をパタンとみなし，プログラムに対するパタンマッチングによって実行が進むパタン駆動による実現を行っている．

システムの記述は，パタンとそれが出現したとき（パタンマッチングが成功したとき）に実行する動作の組を，全てのパタンについて定義することにより行う．この各定義は Miranda の関数定義により次のような形式で表現することができる．

```
function-name pattern arguments = actions
```

関数の左辺に記述するパタン "pattern" は基本的に抽象構文定義の各項目であるから，異なるサブシステムにおいてもそれほど差はない．また定義の項目も共通であることから，プログラムの全体的な構成も似たようなものになる．各サブシステムの特性を決定するのは，関数定義右辺の動作 "action" 部分の記述である．きわめて単純化すれば，ここに値の評価の式を書いたものが実行部であり，型の評価の式を書けば型検査部であるといえる．

以下では，型検査部を例に，より具体的な実現方法を述べる．

3．1　型検査部

型検査部における型検査（あるいは型評価）規則は，各抽象構文パタンごとに，たとえば（Abs_add e1 e2）というパタンに対しては，「e1，e2 が共に整数型のとき，型検査は成功．部分式全体の型として整数型を返す」のように与える．これを Miranda プログラムで記述すると次のようになる．

```
check (Abs_add e1 e2) env
 = Integer_type ,if (check e1 env) = Integer_type &
                    (check e2 env) = Integer_type
 = Error_type   ,otherwise
```

このような関数定義を列挙したシステムに被検査プログラムを与えると，プログラムの構文が再帰的に定義されているため，関数群も再帰的に実行されていき，結局定数および変数の型のレベルから，ボトムアップにプログラム全体の型が決定される．

部分式の型が帰納的に一意に定まるシステムであれば，この方式でほぼ完全な型検査系を構成することができるが，pfp では多相型をサポートしており，型を定数的に扱えない場合が生じる．したがって，型検査規則（型検査アルゴリズム）を拡張する必要がある．今回の型検査部の実現では，型を型変数として表現し，さらにそれに対応して「型の等しさ」の概念を「型変数が単一化可能かどうか」に拡張したアルゴリズム[1]を用いた．

図３．１に今回の実現で用いた型検査アルゴリズムを（自然言語で表現したもの）を示す．ルールを適用すべき抽象構文パタン（１行目，Miranda による抽象構文表現で記述）と，それに対応する型ルール（２行目以降，自然言語で記述）の組を列挙している．ただし，基本型同士の２項演算の場合のような自明なものは省略した．

このアルゴリズムを用いて，式 λx.(λy.xy) の型検査を行った例を以下に示す．なお，どのパタンやルールを適用したかを明確にするために，各パタン／ルール組の先頭の番号を用いて「パタン[20]」「ルール[8]」のように表記した．

(1) 式 λx.(λy.xy) はパタン[28]にマッチするので，対応するルール[28]を適用．x および対応する型変数 σ1を環境に加える．次に λy.xy の評価にうつる．
(2) λy.xy が再びパタン[28]にマッチし，ルール[28]を適用．y と σ2とを環境に加え，xy の評価にうつる．
(3) xy はパタン[10]にマッチ．ルール[10]を適用．ここでまず，ルール[2]および現在の環境より x，y の型はそれぞれ σ1，σ2 となる．新しい型変数 σ3に対して，σ1と（σ2 → σ3）とは単一化可能であるからここでの型検査は成功し，式 xy の型は σ3となる．また，単一化の結果 σ1は（σ2 → σ3）に置き換わる．
(4) (2) におけるルール[28]の続きで，λy.xy の型は

($\sigma 2 \rightarrow \sigma 3$) となる．また，環境から (y, $\sigma 2$) を取り除く．

(5) (1) におけるルール[28]の続きで，$\lambda x.(\lambda y.xy)$ の型は ($\sigma 2 \rightarrow \sigma 3$) $\rightarrow$ ($\sigma 2 \rightarrow \sigma 3$) となる．最後に環境から (x, ($\sigma 2 \rightarrow \sigma 3$)) を取り除く．

図３．２は，図３．１のアルゴリズムを Miranda プログラムとして実現したものの一部である．自然言語による，形式的でないアルゴリズムの記述が，Miranda の記法で比較的素直に実現されており，このような方式が Miranda による開発に適していることがわかる．

なお，各関数の引数としてはパタンの他に，環境（env）および最新の型変数（var）がある（この２つはすべての関数で渡しあっており，事実上共有データとなっている）．環境は上の型検査の実行例の中でも述べたように，識別子とそれに対応する型とからなるテーブルである．一方，このアルゴリズムでは，次々と新しい（今までに使われていない）型変数を作り出す必要があるが，それが真に新しいものであることを保証するために，型変数に順序番号を与えて管理している．「最新の型変数」は次の新しい型変数

[1] Abs_func_def (Abs_binding ident expr)
expr の型を型変数$\tau 1$とし，identと$\tau 1$との対を環境に加える．

[2] Abs_expr expr
この時点の環境を用いて，exprの型検査を行う．

[3] Abs_constant int
部分式の型は，定数 Basic_type_num 型となる．

[4] Abs_expr_id ident
部分式の型は，ident が環境中に存在する場合にはそれに対応する型となり，存在しない場合には新しい型変数$\sigma 1$となる．

[9] Atom expr1
expr1の型検査を行い，その型を$\tau 1$とする．型検査が成功した場合，部分式の型は定数 Basic_type_bool 型となる．

[10] Abs_func_appli expr1 expr2
expr1, expr2 の型をそれぞれ$\tau 1, \tau 2$とし，新しい型変数$\sigma 1$を導入する，$\tau 1$と（$\tau 2 \rightarrow \sigma 1$）が単一化可能ならば型検査は成功し，部分式の型は$\sigma 1$となる．

[16] Abs_add expr1 expr2
expr1, expr2 の型がそれぞれ Basic_type_num 型と単一化可能ならば型検査は成功し，部分式の型は定数 Basic_type_num 型となる．

[27] Abs_if expr1 expr2 expr3
expr1, expr2, expr3 の型をそれぞれ$\tau 1, \tau 2, \tau 3$とすると，$\tau 1$と Basic_type_bool 型が単一化可能，かつ$\tau 2$と$\tau 3$が単一化可能ならば型検査は成功する．$\tau 2$と$\tau 3$の単一化の結果$\tau 4$が部分式の型となる．

[28] Lambda ident expr
ident に対して新しい型変数$\sigma 1$を導入し，これを環境に加える．この環境を用いてexprの型$\tau 1$を決める．その結果，部分式の型は$\sigma 1 \rightarrow \tau 1$となる．最後に，ident とそれに対応する型の対を環境から削除する．

図３．１　型検査アルゴリズム（部分）

```
check_exp        :: expr -> type_env -> type_var -> (pfp_type, type_subs, type_var)

check_exp (Abs_constant int) env var = (Basic_type_num, Null_table, var)        || アルゴリズム [3]

check_exp (Abs_func_appli exp1 exp2) env var            || アルゴリズム [10]
  = (Error, Null_table, var)            ,if (eq_pty pty1 Error) \/ (eq_pty pty2 Error) \/ ~unifiable
  = (apply_type u_subs (Type_variable (Var (n + 1))), compo u_subs (compo subs2 subs1), Var (n + 1))
                                        ,otherwise
    where
    (pty1, subs1, var1) = check_exp exp1 env var
    (pty2, subs2, (Var n)) = check_exp exp2 (apply_env subs1 env) var1
    (unifiable, u_subs) = unify_eq (ls (apply_type subs2 pty1)
                                       (Function_type pty2 (Type_variable (Var (n + 1)))))

check_exp(Abs_if exp1 exp2 exp3) env var                || アルゴリズム [27]
  = (Error, Null_table, var)    ,if (eq_pty pty1 Error) \/ (eq_pty pty2 Error)
                                \/ (eq_pty pty3 Error) \/ ~unifiable
  = (apply_type u_subs pty3, compo u_subs (compo subs3 (compo subs2 subs1)), var3)      ,otherwise
    where
    (pty1, subs1, var1) = check_exp exp1 env var
    (pty2, subs2, var2) = check_exp exp2 (apply_env subs1 env) var1
    (pty3, subs3, var3) = check_exp exp3 (apply_env (compo subs2 subs1) env) var2
    (unifiable, u_subs) = unify_eqs (Lists (ls (apply_type (compo subs3 subs2) pty1) Basic_type_bool)
                                           (Lists (ls (apply_type subs3 pty2) pty3) Empty))

check_exp (Lambda (Abs_id id) exp) env (Var n)          || アルゴリズム [28]
  = (Error, Null_table, (Var n))        ,if eq_pty pty Error
  = (Function_type (apply_type subs (Type_variable (Var (n + 1)))) pty, subs, var1)     ,otherwise
    where
    (pty, subs, var1)
     = check_exp exp (Table (Key id) (Entry (Gen (Type_variable (Var (n + 1))))) env) (Var (n + 1))
```

図３．２　Miranda で記述した型検査プログラム（部分）

を生成するときの順序番号を指示するものである．

3．2　実行部

すでに述べたように，実行部の記述は関数定義の右辺の内容を除けば，型検査部と同様な構成となっている．実行は基本的にはβ簡約に基づいて行うが，識別子の管理のために環境（テーブル）を用いている．

図3．3に Miranda で記述した実行部プログラムの一部を示す．図3．2の型検査プログラムにおいて対応する（同じ構文パタンに対する）関数の記述と比較されたい．

```
eval_expr (Abs_func_appli expl exp2) env
    = eval_expr (beta_reduce explr id exp2) new_env
      where
      (Lambda (Abs_id id) explr, new_env)
          = eval_expr expl env

eval_expr (Abs_if cond then_exp else_exp) env
    = eval_expr then_exp new_env          ,if cond_val
    = eval_expr else_exp new_env          ,otherwise
      where
      (cond_val, new_env) = eval_expr cond env

eval_expr (Lambda id exp) env = (Lambda id exp, env)
```

図3．3　Miranda で記述した実行プログラム

3．3　並列結合子グラフ生成部

2．3節で述べたように，pfp 処理系における並列結合子グラフの生成には，各関数の，引数に対するストリクト性の情報および式の「重み」の評価が必要である．

ストリクト性情報の抽出（ストリクト性解析）には抽象解釈（abstract interpretation）という手法を用いる[7][8]．その詳細はここでは述べないが，これもまた一種の評価機構であり，その実現は構文パタンに対応した評価規則の定義によって行う．つまりこれも型検査部や実行部と同様な形式で記述することになる．

さらに，式の「重み」評価も構文パタンに対する評価規則として定義できるから，プログラムの構成は他のサブシステムとやはり基本的に共通である．なお，現在のところこの評価規則はごく単純なものであるため，実際の記述内容は型検査部に比べてかなり簡単である．

以上のように，抽象プログラムを入力とする型検査部，実行部，並列結合子グラフ生成部は，いづれもパタン駆動手法によって実現することができ，その結果，これらのサブシステムのプログラムは互いに共通な基本的なスタイルを持つことになった．これはシステムの修正や保守においては非常に有利な特性である．

4　一般システムへのパタン駆動手法の適用

言語処理系の場合，取り扱う対象であるプログラムが本質的に規則に基づいて構成された記号列であり，これをパタンの集合としてとらえたシステムを構築することは，比較的自然に行うことができる．本節では，前節で述べたパタン駆動に基づくシステム実現手法を拡張し，より一般的なシステムに適用する方法について述べる．

4．1　パタン駆動手法の拡張

パタン駆動という点からみると，言語処理系における（抽象）構文定義は，処理系が被処理プログラムを走査していく過程で発生（遭遇）する事象の定義であると考えることができる．つまり，抽象プログラムを走査してある抽象構文パタンにマッチしたということを，そのパタンが現れたという事象として解釈するのである．これによって，一般的なシステムについても，そのシステムにおいて発生する事象（システムの状態，システム中でやりとりされる実際のデータなども含む）を，抽象構文定義と同様なデータ構造として定義することによって，パタン駆動手法を用いた記述を行うことができる．

ただし，言語処理系では事象データ構造（被処理プログラム）自身が有限な再帰的構造を持つため，処理関数群もそれに合わせて再帰的に適用されていく．このとき事象の順序関係は問題にならない．また事象データは引数として自然に引き継がれていく．しかし一般には事象データがそのような構造を持つとは限らず，また有限でない場合もありうる．さらに事象間の順序関係が問題になる場合が多い．したがって，関数間の呼び出し関係や事象データの引き渡しの関係などは，データ構造とは別に検討しなければならない．

本方式ではここで，事象の（無限）系列をストリームとし，システムをストリーム処理関数のネットワークとして構成するストリーム計算[9]のアプローチを用いる．この，システム事象のような仮想データからなるストリームを用いたシステムのモデル化は，たとえば Broy によって提案されている[10]が，本方式は，このモデルにおける事象データ構造の定義とストリーム処理関数の構成との間にパタン駆動方式の考え方を導入したものであるともいえる．

4．2　簡単なシステムの記述例

パタン駆動手法によるシステムの記述例として，哲学者の食事問題のシミュレーションをとりあげる．システム構成を図4．1に，また Miranda プログラムリストを図4．2に示す．なおこの例題はすでに文献[11]の中で，ストリーム処理関数のネットワークとして実現されており，今回の実現においてはストリーム処理関数をパタン駆動に基づいて再構成した他は，ほとんど変更はない．

各哲学者は，思索（Thinking），食事の要求（Request），食事（Eating），食事の終了（Release）という事象を繰り返し発生させようとする（ストリーム p0 ～ p4）．manager はフォークの状態および食事を待つ哲学者の状態を見て，次に発生しうる事象を制御するための情報を interleave に送る（ストリーム c）．interleave は，発生可能な事象の中から1つを選んで，それを発生させる

を生成するときの順序番号を指示するものである．

3．2　実行部

すでに述べたように，実行部の記述は関数定義の右辺の内容を除けば，型検査部と同様な構成となっている．実行は基本的にはβ簡約に基づいて行うが，識別子の管理のために環境（テーブル）を用いている．

図3．3に Miranda で記述した実行部プログラムの一部を示す．図3．2の型検査プログラムにおいて対応する（同じ構文パタンに対する）関数の記述と比較されたい．

```
eval_expr (Abs_func_appli expl exp2) env
    = eval_expr (beta_reduce explr id exp2) new_env
      where
      (Lambda (Abs_id id) explr, new_env)
          = eval_expr expl env

eval_expr (Abs_if cond then_exp else_exp) env
    = eval_expr then_exp new_env          ,if cond_val
    = eval_expr else_exp new_env          ,otherwise
      where
      (cond_val, new_env) = eval_expr cond env

eval_expr (Lambda id exp) env = (Lambda id exp, env)
```

図3．3　Miranda で記述した実行プログラム

3．3　並列結合子グラフ生成部

2．3節で述べたように，pfp 処理系における並列結合子グラフの生成には，各関数の，引数に対するストリクト性の情報および式の「重み」の評価が必要である．

ストリクト性情報の抽出（ストリクト性解析）には抽象解釈（abstract interpretation）という手法を用いる[7][8]．その詳細はここでは述べないが，これもまた一種の評価機構であり，その実現は構文パタンに対応した評価規則の定義によって行う．つまりこれも型検査部や実行部と同様な形式で記述することになる．

さらに，式の「重み」評価も構文パタンに対する評価規則として定義できるから，プログラムの構成は他のサブシステムとやはり基本的に共通である．なお，現在のところこの評価規則はごく単純なものであるため，実際の記述内容は型検査部に比べてかなり簡単である．

以上のように，抽象プログラムを入力とする型検査部，実行部，並列結合子グラフ生成部は，いづれもパタン駆動手法によって実現することができ，その結果，これらのサブシステムのプログラムは互いに共通な基本的なスタイルを持つことになった．これはシステムの修正や保守においては非常に有利な特性である．

4　一般システムへのパタン駆動手法の適用

言語処理系の場合，取り扱う対象であるプログラムが本質的に規則に基づいて構成された記号列であり，これをパタンの集合としてとらえたシステムを構築することは，比較的自然に行うことができる．本節では，前節で述べたパタン駆動に基づくシステム実現手法を拡張し，より一般的なシステムに適用する方法について述べる．

4．1　パタン駆動手法の拡張

パタン駆動という点からみると，言語処理系における（抽象）構文定義は，処理系が被処理プログラムを走査していく過程で発生（遭遇）する事象の定義であると考えることができる．つまり，抽象プログラムを走査してある抽象構文パタンにマッチしたということを，そのパタンが現れたという事象として解釈するのである．これによって，一般的なシステムについても，そのシステムにおいて発生する事象（システムの状態，システム中でやりとりされる実際のデータなども含む）を，抽象構文定義と同様なデータ構造として定義することによって，パタン駆動手法を用いた記述を行うことができる．

ただし，言語処理系では事象データ構造（被処理プログラム）自身が有限な再帰的構造を持つため，処理関数群もそれに合わせて再帰的に適用されていく．このとき事象の順序関係は問題にならない．また事象データは引数として自然に引き継がれていく．しかし一般には事象データがそのような構造を持つとは限らず，また有限でない場合もありうる．さらに事象間の順序関係が問題になる場合が多い．したがって，関数間の呼び出し関係や事象データの引き渡しの関係などは，データ構造とは別に検討しなければならない．

本方式ではここで，事象の（無限）系列をストリームとし，システムをストリーム処理関数のネットワークとして構成するストリーム計算[9]のアプローチを用いる．この，システム事象のような仮想データからなるストリームを用いたシステムのモデル化は，たとえば Broy によって提案されている[10]が，本方式は，このモデルにおける事象データ構造の定義とストリーム処理関数の構成との間にパタン駆動方式の考え方を導入したものであるともいえる．

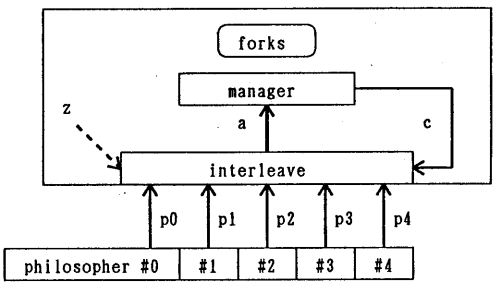

図4．1　哲学者の食事問題のシステム構成
（文献[11]による）

4.2 簡単なシステムの記述例

パタン駆動手法によるシステムの記述例として，哲学者の食事問題のシミュレーションをとりあげる．システム構成を図4.1に，また Miranda プログラムリストを図4.2に示す．なおこの例題はすでに文献[11]の中で，ストリーム処理関数のネットワークとして実現されており，今回の実現においてはストリーム処理関数をパタン駆動に基づいて再構成した他は，ほとんど変更はない．

各哲学者は，思索（Thinking），食事の要求（Request），食事（Eating），食事の終了（Release）という事象を繰り返し発生させようとする（ストリーム p0 ～ p4）．manager はフォークの状態および食事を待つ哲学者の状態

```
|| actions of philosophers

action ::= Thinking num | Request num
         | Eating num | Release num

|| control information from manager

control ::= SIG num | BLK ([num],[num])

|| forks

fork ::= INITIAL | GRASP fork num

grasp f n = GRASP f n
release (GRASP f j) i
    = f                 ,if i=j
    = grasp INITIAL j   ,if f=GRASP INITIAL i

isok INITIAL i = True
isok (GRASP INITIAL j) i
    = False, i=(right j) \/ i=(left j)
    = True, otherwise
isok (GRASP (GRASP INITIAL j) k) i = False

|| philosopher

philosopher :: num -> [action]
philosopher i
    = Thinking i : Request i : Eating i : Release i
      : philosopher i

|| manager

manager :: fork -> [action] -> ([num],[num]) -> [control]

manager f [] q = []

manager f ((Thinking i):e) q = BLK q : manager f e q

manager f ((Request i):e) q
    = BLK (insertpq i q) : manager f e (insertpq i q)
              ,if ~isok f i
    = BLK (insertpq i q) : manager f e (insertpq i q)
              ,if isinhighpq (left i) q \/
                  isinhighpq (right i) q
    = SIG i : manager (grasp f i) e q
              ,otherwise

manager f ((Eating i):e) q = BLK q : manager f e q

manager f ((Release i):e) q
    = SIG (right i) :
      man2 (grasp (release f i) (right i)) e
           (removepq (right i) q) i
              ,if isinhighpq (right i) q &
                  isok (release f i) (right i)
    = man2 (release f i) ((Release i):e) q i
              ,otherwise

man2 f e q w
    = SIG (left w) :
      manager (grasp f (left w)) (tl e) (removepq (left w) q)
              ,if isinhighpq (left w) q & isok f (left w)
    = BLK q : manager f (tl e) q
              ,otherwise

right i = (i-1) mod 5
left  i = (i+1) mod 5
```

```
|| interleave

interleave z c [] [] [] [] [] = []

interleave z ((SIG 0):c) a0 a1 a2 a3 a4
    = hd a0 : interleave z c (tl a0) a1 a2 a3 a4

interleave z ((SIG 1):c) a0 a1 a2 a3 a4
    = hd a1 : interleave z c a0 (tl a1) a2 a3 a4

interleave z ((SIG 2):c) a0 a1 a2 a3 a4
    = hd a2 : interleave z c a0 a1 (tl a2) a3 a4

interleave z ((SIG 3):c) a0 a1 a2 a3 a4
    = hd a3 : interleave z c a0 a1 a2 (tl a3) a4

interleave z ((SIG 4):c) a0 a1 a2 a3 a4
    = hd a4 : interleave z c a0 a1 a2 a3 (tl a4)

interleave z ((BLK (q1, q2)):c) a0 a1 a2 a3 a4
    = hd a0 : interleave (tl z) c (tl a0) a1 a2 a3 a4
                ,if z=0 & a0~=[] & ~(isin 0 (q1++q2))
    = hd a1 : interleave (tl z) c a0 (tl a1) a2 a3 a4
                ,if z=1 & a1~=[] & ~(isin 1 (q1++q2))
    = hd a2 : interleave (tl z) c a0 a1 (tl a2) a3 a4
                ,if z=2 & a2~=[] & ~(isin 2 (q1++q2))
    = hd a3 : interleave (tl z) c a0 a1 a2 (tl a3) a4
                ,if z=3 & a3~=[] & ~(isin 3 (q1++q2))
    = hd a4 : interleave (tl z) c a0 a1 a2 a3 (tl a4)
                ,if z=4 & a4~=[] & ~(isin 4 (q1++q2))
    = interleave (tl z) ((BLK (q1, q2)):c) a0 a1 a2 a3 a4
                ,otherwise

|| queue

insert i q = i : q

remove i q = [], q=[]
           = tl q, hd q=i
           = hd q : remove i (tl q), otherwise

isin i s = False, s=[]
         = True, i=hd s
         = isin i (tl s), otherwise

isinhighpq i (q1,q2) = isin i q1

insertpq i (q1,q2)
    = (q1, (insert i q2)), isin (left i) q1 \/ isin (right i) q1
    = ((insert i q1), q2), otherwise

removepq i (q1,q2)
    = ((insert(left i)(insert(right i)(remove i q1))),
       (remove(left i)(remove(right i) q2)))
                ,if isin (left i) q2 & isin (right i) q2
    = ((insert (left i) (remove i q1)), (remove (left i) q2))
                ,if isin (left i) q2
    = ((insert (right i) (remove i q1)), (remove (right i) q2))
                ,if isin (right i) q2
    = ((remove i q1), q2)
                ,otherwise

|| system configuration

a = interleave z c p0 p1 p2 p3 p4
c = BLK ([],[]) : manager INITIAL a ([],[])
p0 = philosopher 0
p1 = philosopher 1
p2 = philosopher 2
p3 = philosopher 3
p4 = philosopher 4
```

図4.2　哲学者の食事問題の Miranda プログラム

を見て，次に発生しうる事象を制御するための情報を interleave に送る（ストリーム c）．interleave は，発生可能な事象の中から１つを選んで，それを発生させる（ストリーム a）．どれを選ぶかは interleave へのストリーム z によって指定する．

ストリームは Miranda のリスト型として表現する．遅延評価のため，こうした無限の系列も自然に表現することができる．

このシステムで事象データ構造として扱うのは，哲学者が発生する事象（action 型）および manager が出力する制御情報（control 型）で，ともに代数データ型で定義している．パタン駆動方式で構成した関数は manager および interleave で，それぞれストリーム a，ストリーム c をパタンとみなす．

4.3 考察

現在までに哲学者の食事問題を含め，２，３の簡単な例題についてパタン駆動の手法を適用してみた．これまでに明らかになった，あるいは予想できる，方式の利点および問題点には次のようなものがある．

(1) pfp 処理系の場合と同様に，全体として見通しのよい記述が可能．
(2) 適切な事象データ構造とストリーム関数ネットワークの設定が難しい．
(3) 関数が取り扱う条件が複雑になると，パタンマッチングだけで定義できる範囲が狭くなる．
(4) 冗長な記述が増加する（適切な条件節を用いた記述に比べて）．ただし，これは (1) の見通しのよさとのトレードオフでもあり，必ずしも欠点とは言えない．

この他にも細かい問題点はいくつか指摘できるが，その多くは（上に挙げたものも含めて）(2) の，事象データ構造および関数構成の設定をどのようにするかという問題に帰着する．この部分を定式化することが，パタン駆動アプローチを一般的なシステム記述に適用していく上での最大の課題であろう．

5 まとめ

今回パタン駆動アプローチを用いて，言語処理系と，哲学者の食事問題のプログラムを構成した．言語処理については，処理対象となるプログラム列が本来パタンとしての取り扱いがしやすいこともあり，非常に自然で簡潔なシステムの記述を行うことができた．

一方，より一般的な問題に対するパタン駆動アプローチの適用は，現在のところまだ言語処理系の場合のようなシステムを完全に記述するための手法としては問題点も多いが，１つのプログラミングスタイルとして部分的に適用する場合には有効性が期待できる．

現在，パタン駆動手法の応用事例として，pfp の並列簡約系のシミュレータの実現を行っている．今後さらに，こうした実用規模の問題に本手法の適用を試み，問題点の改善を行っていく．

参考文献

[1] Bergstra, J.A., Heering, J. and Klint, P. (eds.): Algebraic Specification, Addison-Weslay, 1989

[2] 大黒，荒木：関数型プログラミング言語 pfp の設計と実現，九州大学工学集報 Vol.61, No.5, pp647-654, 1988

[3] Turner, D.A.: Miranda: A Non-strict Functional Language with Polymophic Types, LNCS Vol.201, Springer-Verlag, pp1-16, 1985

[4] Miranda: Miranda System Manual, Research Software Limited, 1989

[5] 堀，広川，関本：結合子を用いた並列リダクション，電子情報通信学会論文誌Ｄ，Vol.J70-D, No.8, pp1498-1507, 1987

[6] 中山：結合子を用いた並列実行制御方式，日本ソフトウェア科学会第６回大会論文集，pp401-404, 1989

[7] Clack, C. and Peyton Jones, S.L.: Strictness Analysis - a practicalapproach, LNCS Vol.201, Springer-Verlag, pp35-49, 1985

[8] Peyton Jones, S.L. et al.: The Implementation of Functional Programming Languages, Prentice-Hall International, 1987

[9] 田中：関数型プログラムにおけるストリーム計算，情報処理，Vol.29, No.8, pp836-844, 1988

[10] Broy, M.: Requirement and Design Specification for Distributed Systems, Proc. CONCURRENCY 88, LNCS Vol.335, Springer-Verlag, pp33-62, 1988

[11] 荒木：並行動作システムの Miranda による仕様記述，コンピュータソフトウェア，Vol.8, No.1, pp12-24, 1991