ソフトウェア基礎論　37－４
プログラミング言語　27－４
（1990. 12. 13）

関数型言語を用いた
並列関数型言語処理系開発のための一手法

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

あらまし　筆者らは現在，結合子グラフ簡約に基づく関数型言語の並列実行方式に関する研究を行っている．今回，この研究の一環として，並列実行系の検討や評価を行うための実験環境を用意するため，簡単な関数型言語の処理系および実行系の試作を行う．このシステムは原始プログラムから中間形式（抽象構文木）への変換部（パーザ）と，実行部（逐次処理），型検査部，および並列結合子グラフ生成部からなり，これらをすべて関数型言語を用いて記述する．パーザ部を除く各部は，抽象構文パタンに対するアクション（ルール）の集合という形式で記述し，パタン駆動による動作を行う．このパタン／ルールを列挙する記述形式は，関数型言語のプログラムで自然に表現することができ，見通しのよいシステム記述が可能である．

A Technique for Developing Parallel Functional Language Systems Using Functional Languages

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

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

Abstract　We are studying a parallel execution method of functional languages based on the combinator graph reduction. We are building a simple functional language system to provide a experimenting environment for test and evaluation of our parallel execution method. This system consists of parser, executor (sequential), type checker and parallel combinator graph generator. The parser translates source programs into intermediate form (abstract syntax tree). The whole system are written in another functional language. All the subsystems except for the parser are organized with an uniform methodology based on pattern driven. This system describing method can be implemented naturaly using a functional language.

1　はじめに

関数型プログラムは，比較的容易に並列性を抽出し，利用することができるため，多くの並列実行方式が提案，研究されている．筆者らは比較的低並列度のマルチプロセサあるいは分散処理環境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，その研究を行っている[6]．この並列実行方式を具体化し，検討や評価を行うための実験環境が必要となったため，今回，pfp[2]という比較的簡単な関数型言語を取り上げ，その処理系を作製することにした．

この処理系はあくまで実行方式研究のための道具であるから，短期間で容易に開発できて，しかも利用を始めた後も，各種の仕様変更に対応しやすいものであることが望ましい．そのため，

- 高水準（形式的仕様記述かそれに近いレベル）でシステムを記述し，開発コストの低減をはかる．
- 処理系各部をモジュール化し，独立性を高める．
- 仕様の変更をシステムに反映させやすいような設計を行う．
- 当面，実行効率や性能は問わない．

等の設計方針を設定した．

これらの方針（特に高水準の記述）に合致するシステム記述の方法としては，たとえば形式的仕様記述言語を用いることが考えられる．しかし，現在提案されている仕様記述言語の多くは，システムの振る舞いを形式的に取り扱うための仕様記法を提供するものであり，それ自身が実際に動作するというものではない．

次に関数型言語のような，問題とその解法を抽象度の高いまま記述できるようなプログラミング言語の利用が考えられる．たとえば，関数の定義の集積によってプログラムを構成する関数型言語は，等式の集合からなる代数的仕様記述の表現を，ほとんどそのままの形式で記述することができ，しかも，その記述はプログラムとして動作する．すなわち，関数型言語を用いることにより，実際に動作するシステムを，仕様記述に近いレベルで記述することが可能となる．

本研究では，処理系の記述と実現に関数型言語 Miranda[3][4]を使用し，処理系の開発とともに，関数型言語による実用規模のシステム記述の事例研究を行うことにした．実際の開発作業を通じて，こうした言語処理系やさらに大規模なシステム記述に関数型言語を適用する場合の方法論の開発や，あるいは既存の C 言語などによるシステム開発と比較した，関数型言語による開発の利点と問題点の検討などを行う．

```
 We shall use variables to range over the various
constructs of pfp as follows:

          P  ranges over pfp programs
          p  ranges over pfp program constructors
          d  ranges over pfp definitions
          d  ranges over pfp bindings
          v  ranges over pfp varstructs
                             (variable structures)
          e  ranges over pfp expressions
          c  ranges over pfp constants
          id ranges over pfp identifiers

Syntax equations for pfp

P  ::= p1; p2; ... pn;               pfp program

p  ::= d                             function definition
     | e                             expression

d  ::= def b                         recursive definition

b  ::= id = e                        simple binding
     | id v1 v2 .. vn = e            function binding

v  ::= id                            variable
     | (id1, id2, ..., idn)          n-tuple

e  ::= c                             constant
     | id                            identifier

     | ~ e                           negation
     | hd e                          list head
     | tl e                          list tail
     | null e                        null check
     | atom e                        atom check

     | e1 e2               L         function application
     | - e                           unary minus

     | e1 * e2             L         multiplication
     | e1 / e2             L         division
     | e1 % e2             L         modulo
     | e1 & e2             L         conjunction

     | e1 + e2             L         addition
     | e1 - e2             L         subtraction
     | e1 | e2             L         disjunction

     | e1 : e2             R         list construction
     | e1 ^ e2             R         list append

     | e1 = e2             L         atomic equality
     | e1 ~= e2            L         atomic inequarity
     | e1 < e2             L         less than
     | e1 > e2             L         greater than
     | e1 <= e2            L      less than or equal to
     | e1 >= e2            L      greater than or equal to

     | if e1 then e2 else e3         conditional

     | lambda v1 v2 .. vn.e          lambda abstraction

     | []                            empty list
     | [e1, e2, ..., en]             list of n elements

     | (e1, e2, ..., en)             n-tuple

     | (e)                           equivalent to e

Notes:
1. In the syntax equations, constructs are listed in
   order of decreasing binding power. For example,
   since "e1 e2" is listed before "e1 + e2",
   function application binds tighter than addition,
   so "e1 e2 + e3" is parsed as "(e1 e2) + e3".
2. "L" or "R" means that it associates to the left(L)
   or right(R) when juxtaposed with itself.
   For example, "e1 e2 e3" is parsed as "(e1 e2) e3".
```

図2．1　pfp の構文

本稿では，まず２章で，今回実現する pfp と，その記述に用いる Miranda との２つの関数型言語について簡単に紹介する．３章では処理系の全体的な構成について説明し，４章でそのうちの型検査部と実行部の具体的な実現技法について述べる．５章では，処理系本来の目的である並列実行系の実現方式の概要を説明し，最後に６章では今回用いた実現方式の評価，および関数型言語を用いたシステム開発の問題点等について述べる．

2　関数型言語 pfp と Miranda

今回処理系を作成する pfp 言語[2]は，比較的小規模な言語仕様をもつ関数型言語であり，次のような特徴を持つ．

- 高階関数，多相型，型変数のサポート
- 強い型付け
- 遅延評価の採用

pfp の構文を図２．１に示す．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基本演算（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，実験用言語は実現が容易であると同時に，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 の言語仕様は，このような点からみて十分なものであると判断し，今回取り上げることにした．

一方，記述言語として用いる関数型言語 Miranda[3][4]は，上で述べた pfp 言語の機能をほぼ包含し，さらに，

- 代数的データ型，抽象データ型，型変数など，より柔軟な型システム
- パタンマッチング

などの機能を備えている．今回の実現では特にこのパタンマッチングの機能を多用した．また実用的な処理系として，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なども持っているため，今回のような実用システムの構築にも利用しやすい．

3　システムの概要

図３．１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．字句解析，構文解析とも，構文パタンと，それに対応する構文木の生成規則の組を用意し，パタンマッチングによって構文木の組み立てを行っていく方式[1]を用いている．

pfp の抽象構文は Miranda のデータ構造（代数的データ型）を用いて図３．２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（図２．１）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３．３(a) の pfp プログラムを抽象構文表現に変換したものを図３．３(b)に示す．

システムの残りの部分，すなわち型検査部，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．

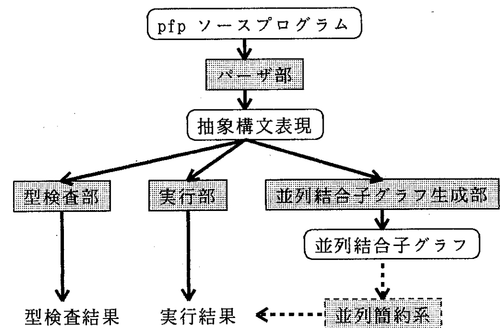

図３．１　システム構成

4　型検査部および実行部

本処理系では，型検査部，実行部とも，抽象構文プログラムをパタンとするパタン駆動に基づく実現を行った．システムの記述は，抽象構文定義（図３．２）の各項目ごとに，それをパタンとして左辺におき，対応する動作を右辺に記述した（Miranda の）関数定義（以下パタン関数と呼ぶ）を列挙することにより行う．このシステムに抽象構文プログラム（パタン）を入力すると，パタンマッチしたパタン関数が順次実行され，処理が行われる．

この方式では，各パタン関数ごとの処理が比較的独立しているため，システムの記述をパタン関数のレベルで分割して考えることができる．そのため，システム開発が容易となり，また信頼性も向上する．また，型検査部，実行部の全体的な構成が共通となるため，システム全体の把握もしやすい．

4．1　型検査部

型検査系におけるパタン関数の右辺には型検査ルールを記述する．たとえば，(Abs_add e1 e2) という抽象構文パ

```
||***** 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３．２　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３．３　抽象構文表現の例

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」という型検査ルールを与えることになる．Miranda プログラムでは次のようになる．

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

このシステムに型検査を行う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，パタン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，ルール（型検査アルゴリズム）を拡張する必要がある．そこで，今回の型検査系の実現では，型を型変数として表現することでそうした問題に対処し，さらにそれに伴って「型の等しさ」の概念を「型変数が単一化可能かどうか」に拡張したアプローチ[1]を用いた．

図４．１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$\sigma 1$は（$\sigma 2 \rightarrow \sigma 3$）に置き換わる．

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

図4．2は，図4．1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

[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4．1　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 (Is (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 (Is (apply_type (compo subs3 subs2) pty1) Basic_type_bool)
                                       (Lists (Is (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4．2　Miranda で記述した型検査プログラム（部分）

えて管理している．「最新の型変数」は次の新しい型変数を生成するときの順序番号を指示するものである．

4．2　実行部

前にも述べたとおり，実行部の基本的な構成もまた，型検査部のそれと同様に，構文パタンと対応する動作の記述（パタン関数）によって行う．パタンの集合は共通であるから，両者の違いの大部分は動作の定義（関数の右辺）部にある．型検査部における動作は，主に，型の条件の検査および型変数の単一化であった．実行部においては，これが式の評価になっている．

今回の処理系は，基本的には β 簡約に基づく簡約実行系として実現しているが，識別子（主に関数名）の管理はテーブルによって行う．

図4．3に Miranda で記述した実行部プログラムの一部を示す．

```
eval_expr (Abs_func_appli exp1 exp2) env
    = eval_expr (beta_reduce exp1r id exp2) new_env
      where
      (Lambda (Abs_id id) exp1r, new_env)
          = eval_expr exp1 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4．3　Miranda で記述した実行プログラム

5　並列結合子グラフ生成部

本章では，この処理系作成の主目的である，並列実行方式の実験システムの実現について述べる．ただしここで説明するのは，本方式の並列実行系においての実行形式プログラムである並列結合子グラフを生成する部分である．並列実行系自体は，UNIX プロセスを用いた疑似並列環境または分散環境上に，C 言語等を用いて実現する予定なので，本稿では省略する．

5．1　並列結合子簡約方式の概要

今回実現を試みる並列結合子簡約系[6]は，並列制御結合子（PRCC）を用いた並列簡約方式を基にして，並列度の低いハードウェア環境での実行効率の改善を図ったものである．

PRCC[5]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，後者はストリクト性解析を用いて抽出する．

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．今回はごく粗い近似として，5．4節で述べるような簡易な方法をとった．

5．2　並列結合子グラフ生成部の概要

本システムにおける並列結合子生成部の概要を，図5．1に示す．

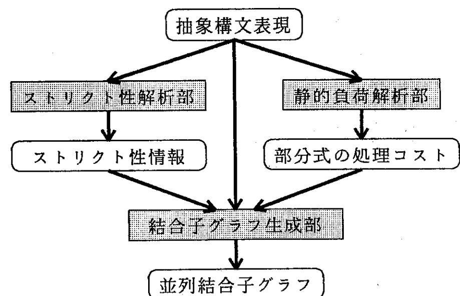

図5．1　並列結合子グラフ生成部の概要

前節で述べたように，今回実現しようとしている並列実行系では，PRCC の生成のためにストリクト性の情報，また並列処理単位の切り分けのために実行コストの評価を必要とする．ストリクト性解析部および静的負荷解析部はそれぞれ，抽象構文表現されたプログラムからそれらの情報を抽出し，生成部に渡す．生成部はこれらの情報を用いて，抽象構文表現プログラムから結合子式を生成する．

これらのサブシステムについても，前章の型検査や実行系と同様なパタン駆動に基づく実現を行う．

5．3　ストリクト性解析部

ストリクト性解析[7][8]は，引数をとる式に対して，その引数が必須のものであるかどうか，つまりその式の値を決定するためにその引数の値が必要かどうかを調べるものである．並列実行においてストリクト性の情報は，たとえば引数が必須であれば式の評価と引数の評価を並列に行ってもよいが，そうでない場合に両者を並列に評価しようとすると，引数の計算が無駄になる可能性がある，といったことをチェックするのに利用することができる．本システムでは，PRCC に付与する実行制御情報を得るために利用する．

形式的には，式 f がその引数についてストリクトかどうかは，その引数として⊥（bottom: non-terminating）を与えた結果により決定する．すなわち，

$$f \perp = \perp$$

ならば，f はその引数においてストリクトであり，そうでなければストリクトではない．

さて，具体的にストリクト性解析を行うためには，抽象解釈（abstract interpretation）という手法を用いる．これは，

$$\text{abs } \perp = 0$$
$$\text{abs } x = 1 \qquad (x \neq \perp)$$

なる写像 abs を考え，この写像の世界で式の値を求めるというものである．たとえば，いま abs で写像した世界において式 f に対応するもの（抽象化された f）を f# とすると，f# 0 の値が 0 であればストリクトであり，そうでなければストリクトではないことになる．基本的な演算等について抽象化してみると，加算（x + y）は２つの引数が両方とも停止しないと停止しないので，抽象化した結果は (x +# y) = (x & y)（& は論理積）となる．同様に if は，(if# c t e) = c & (t | e)（| は論理和）となる．このような規則を組み合わせることにより，任意の式 f の抽象化を得ることができる．たとえば，

```
f p q r = if (= p 0) (+ q r) (+ q p)
```

という式の抽象化は

```
f# p q r = if# (=# p 0) (+# q r) (+# q p)
         = & (& p 1) (| (& q r) (& q p))
         = & p (& q (| p r))
```

となる．そこで各引数についてのストリクト性を調べてみると，

```
f# 0 1 1 = & 0 (& q (| p r)) = 0
f# 1 0 1 = 0
f# 1 1 0 = 1
```

となり，f は p, q についてはストリクト，r についてはストリクトではないことがわかる．

このストリクト性解析の方法も，前章のパタン関数を使った手法で実現することが可能である．つまり，パタンに対する動作として，そのパタンが示す構文が抽象化した場合の演算規則を記述すればよい．ただ，型検査や実行においては，最終的な型検査結果や実行結果のみが必要であったのに対し，ストリクト性解析では各部分式ごとの結果が必要である．そのため，ここでは抽象構文木と対応付けができる形式の新たなデータ構造を用意して，解析結果を格納する．

5．4　静的負荷解析部

ここでは，プログラムの各部分式の実行コストを実行前に静的に見積もる作業を行う．実行コストの見積もりとはいっても，プログラムを，あまり処理の規模が片寄らない程度に複数の並列処理単位に分割するための根拠が得られればよいので，次のような非常に単純な方法で行っている．

プログラムを２進の構文木の形で考えたときに，その葉の位置に存在する基本処理に対して，それぞれ適当な重み値を与える．その後木を根の方向にたどりながら，すべての節点に，子供の節点の重み値の和を，その節点の重み値として与えていく．

この負荷解析のアルゴリズムの精密化と，それがシステムに与える影響の評価は今後の課題であるが，一方，現在の方法はきわめて単純なものであるから，たとえばパーザに組み込む（原始プログラムを走査するときに同時に解析を済ませてしまう）ことによって，より効率的に解析を行うことも可能である．

5．5　結合子グラフ生成部

結合子グラフ生成部では，基本的には PRCC 生成アルゴリズムを実行して，抽象構文表現プログラムを PRCC グラフに変換していく（この際ストリクト性についての情報が用いられる）．ただそのままでは完全な PRCC グラフに変換されてしまうので，この変換過程を適当なところで打ち切る．このとき PRCC に変換されずに残っているλ式（スーパーコンビネータ）を並列処理単位とする．

この変換打ち切りの条件として５．４節の重み値を用いる．すなわち，部分式の変換を進めていって，残りの部分式につく重み値がある一定値を下回ったら，その部分式についてはそれ以上の変換を行わず，１つの並列処理単位として扱うことにする．

6　まとめ

今回の実現では，抽象構文表現というデータ構造をまず定義し，それを取り扱う各サブシステムを設計した．このとき，データ構造をパタンとし，サブシステムをパタン駆動に基づいて動作するという，これまでに述べたような構成方針をとることで，システム全体として，抽象構文表現のデータ構造を強く反映した，共通の構造を持つことができた．

このため，抽象構文の定義を把握しておけば，サブシステムの記述を追うことは比較的容易であり，システムの可読性は高い．おそらく，言語仕様の変更（つまり抽象構文定義の変更）をシステムに反映させることも容易であろう．

また，パタンと，それに対応する動作を関数としてまとめ，これを列挙することによってシステムを構成する方式は，関数型言語における関数の本質，「左辺から右辺への書き換え規則」，とも親和性が高く，Miranda によって自然に実現することができた．さらに，パタンに対する関数は他とほとんど独立に記述することができるため，問題（システム）をパタン関数の単位まで分割して解決することができた．

システムの構造をうまく反映するデータ構造（これも一種の仕様の表現であろう）が設定できれば，今回のようなパタンベースによるシステム記述手法は，関数型言語を用いたシステム記述の方法論として有効であると言える．

今後の作業としては，疑似並列環境あるいは分散環境上で，本システムが出力する結合子グラフを簡約する並列簡約系を開発し，並列実行方式の実験，評価を行う予定である．また，今回の手法で別のシステムを構築してみて，さらに本手法の評価を行いたいと考えている．

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] Hudak, P. and Young, J.: Higher-Order Strictness Analysis in Untyped Lambda Calculus, Proc. of the 12th ACM Symp. on Principles of Programming Languages, pp97-108, 1986

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