ソフトウェア基礎論　36－4
（１９９０．　９．　２１）

分散型関数型言語Ｃｍｅｘの開発

藤本 聖　白川 洋充　大野 豊

立命館大学　理工学部

分散型関数型言語Ｃｍｅｘ(Concurrent Manipulation of EXpressions) は，Miranda をベース言語とした関数型言語である．Ｃｍｅｘは分散OS上で実現されている．Ｃｍｅｘは Miranda にプロセスの概念を導入し，並列処理の記述を可能にしている．プロセスは，評価の過程で動的に生成される．また，プロセス間の通信を行うためにポートを導入している．ユーザはこのポートを用いてプロセス間通信を記述し，複数のプロセスの協調動作による並列処理を行うことが可能である．

本文では，ベース言語について述べた後，Ｃｍｅｘの基本機構を示し，その実現機構とＣｍｅｘを用いた並列処理記述の適用例を示す．

Development of a Distributed Functional Programming Language Cmex

Satoshi Fujimoto　Hiromitsu Shirakawa　Yutaka Ohno

Ritsumeikan University　Faculty of Science and Engineering

Cmex (Concurrent Manipulation of EXpressions) is the name of distributed functional programming language based on the functional language Miranda. Cmex is implemented on a distributed operating system. Cmex makes it possible to describe parallelism by introducing the concept of process to Miranda. The processes are created dynamically while evaluation is performed. Cmex introduces the port for communicating between the processes. User may describe co-operating processes utilizing a interprocess communication through the ports.

In this paper, we introduce the base language of Cmex and show the basic concepts of Cmex. Also we show a realization method of Cmex and examples of Cmex which are applied to concurrent problems.

1. はじめに

近年のコンピュータの並列処理技術の進歩に伴い，そのハードウェア上で動作する並列処理言語に関する研究も多く行われるようになってきている．我々は，関数型言語をベースとした並列処理言語の開発を行っている．純粋な関数型言語は副作用を持たず，与えられた引数のみから結果を求める．したがって並列処理の記述において，他の部分の処理を意識することなく関数を記述することが可能である．また，すべてのオブジェクトがファーストクラスであるため関数を他の関数の引数として与えることが可能であり，分散環境での評価も容易である[1) 2) 3)]．これらの関数型言語の利点を用いることによって，一般的に非常に複雑になりやすい並列処理の記述をより簡潔にできると考えられる．

分散型関数型言語Ｃｍｅｘ(Concurrent Manipulation of EXpressions)は，Miranda[4)]をベース言語とした，分散OS上で動作する関数型言語である．　Ｃｍｅｘは，Miranda にプロセスの概念を導入し，並列処理の記述を可能にしている．プロセスは並列処理の単位であり，評価の過程で動的に生成される．また，プロセス間の通信を行うためにポートを導入している．ユーザはこのポートを用いてプロセス間通信を記述し，複数のプロセスの協調動作による並列処理を行うことが可能である．

本論文では，ベース言語について述べた後，Ｃｍｅｘの基本機構を示し，その実現機構と，Ｃｍｅｘを用いた並列処理記述の応用例を示す．

2. ベース言語

2.1 Miranda

Mirandaは，D. A. Turner によって開発された関数型言語であり，次のような特徴を持つ．

- ノンストリクト性
- ガード式とパターンマッチングによる簡潔な記述法
- ドット式とＺＦ式を用いたリストの簡便な取り扱い
- 型推論機構による強力な型付け

Ｃｍｅｘでは，浮動小数点数，UNIX関連コマンド以外の Miranda のすべての機能をサポートしている．　図1 に基本演算子を示す．

(1) 算術
　　+, -, *, /, ^, mod
(2) 比較
　　>, >=, =, ~=, <=, <
(3) 論理
　　& (AND), ¥/ (OR), ~ (NOT)
(4) リスト
　　: (コンストラクタ), # (長さ),
　　! (要素選択), ++ (結合), -- (差分)
(5) その他
　　. (関数合成)

図1　基本演算子

2.2 関数定義

プログラムは関数とデータオブジェクトの定義式の集合であり，スクリプトと呼ぶ．where 句を用いることにより，関数の定義

式の中で局所的な関数やオブジェクトの定義を行うことができる．

ローカル変数や関数の定義のスコープを示すために、オフサイドを用いたインデントルール[5] を用いている．前行の先頭より右側から始まる行は前行の継続行であり，同じかそれよりも左から始まる行は，前行と独立しているとみなされる．図2 に関数定義の例を示す．

```
dist a b c = sq b - 4 * a * c
               where
                 sq n = n * n
```

図2　関数の定義

2.3 ガード式とパターンマッチング

場合分けを処理するためにガード式と関数の引数によるパターンマッチングを用いる．ガード式は "式, 条件" の集合であり，条件が満たされたとき，前にある式の評価を行う．図3 に絶対値を求める関数 abs のガード式を用いた定義を示す。

```
abs n = -n ,  n < 0
      = n ,  otherwise
```

図3　ガード式を用いた関数の定義

パターンマッチングは，関数の引数部分にパターンを記述し，そのパターンによって定義式を書き分けるものであり，このパターンマッチングを用いることによってリスト処理等の関数を簡潔に記述することができる．図4 にパターンマッチングを用いた関数の定義例を示す。

```
last [m]    = m
last (m:lm) = last lm

fib 0       = 1
fib 1       = 1
fib (n+2)   = fib (n+1) + fib n

fst (a,b) = a
snd (a,b) = b
```

図4　パターンマッチングを用いた関数の定義

2.4 ドット式とZF式

複雑な要素を持つリストを簡潔に記述する表記法にドット式とZF式がある．ドット式は等差数列のリストを作るのに用いる．書式は [初期値，公差 .. 終値] であり，公差と終値は省略が可能である．省略された場合、公差は 1，終値は ∞であるとみなされる。図5 にドット式の例を示す。

```
even = [0, 2 .. 10]   || is [0, 2, 4, 6, 8, 10]
natural = [0 .. ]     || is [0, 1, 2, 3, ···
```

図5　ドット式

ZF式は，数学における内包的記法を用いてリストの要素を表現するために用いる．例えば，100 以内の整数で 5 で割った余りが 3 である数の集合(リスト)は，図6 のように記述できる．

```
[n ¦ n <- [1..100]; n mod 5 = 3]
```

図6　ZF 式

2.5　型

2.5.1　型推論機構と基本型

言語は，強力に型付けされている．すなわち，スクリプトの中のすべての関数やオブジェクトの型はコンパイル時に静的に決定される．しかし，ユーザは必ずしも関数やオブジェクトの型を宣言する必要はない．型が宣言されていない関数やオブジェクトは，スクリプト内の情報から，Milnerの方法[6] にもとづく型推論機構によって型が推論される．図 7 に基本の型を示す．

名称	表記	意味
整数型	num	整数
文字型	char	文字
論理型	bool	論理値
リスト型	[t]	型 t の要素のリスト
タプル型	(t1,t2,..)	型 t1，t2，...を要素とするタプル
関数型	t1 -> t2	引数の型 t1，返値の型 t2 の関数

図7　基本型

型の宣言は，"::" を用いて行う．また，型変数 *, **, *** を用いることにより任意の型を表現することができる．図8 に型の宣言の例を示す。

```
even :: [num]
abs  :: num -> num
fib  :: num -> num
last :: [*] -> *
fst  :: (*,**) -> *
snd  :: (*,**) -> **
```

図8　型の宣言

2.5.2　型の定義

ユーザは次の三つの型を定義することができる

(1) 型類義
(2) 代数データ型
(3) 抽象データ型

(1) 型類義

型類義は，すでに存在している型に別名をつけることであり，"==" を用いて定義する．図9 に型類義の例を示す。

```
string     == [char]
num_to_str == num -> string
```

図9　型類義

図9 では，string を char のリスト型に，num_to_str を引数が num 型，返値が char のリスト型である関数の型に定義している．

(2) 代数データ型

ユーザは，構築子を用いて新しい型（代数データ型）を定義することができる．構築子は，英大文字から始まる識別子であり，

0 個以上の引数をとる．代数データ型の定義には "::=" を用いる．図10 に代数型の定義とそれを用いた関数の定義の例を示す．

```
day ::= Morning ¦ Noon ¦ Evening ¦Night

tree * :: = TREE (tree *) (tree *) ¦ LEAF *

sum_t :: tree num -> num
sum_t (TREE m n) = sum_t m + sum_t n
sum_t (LEAF m)   = m
```

図10　代数データ型

(3) 抽象データ型

抽象データ型は，データ型とそれを扱う操作関数をひとまとめにしたものである．抽象データ型は abstype によって宣言される．そして， with 以下にそのデータ型を操作する関数の型宣言を記述する．この with 部分に記述された関数以外は， このデータ型を操作することはできない．図11に抽象データ型を用いたスタック型とスタック型を扱う関数の定義の例を示す。

```
abstype  stack *
with empty   :: stack *
     isempty :: stack * -> bool
     push    :: * -> stack * -> stack *
     pop     :: stack * -> stack *
     top     :: stack * -> *

stack * == [*]
empty     = []
isempty x = (x = [])
push a x  = a : x
pop (a:x) = x
top (a:x) = a
```

図11　抽象データ型

3. 基本機構

3.1 preset 句

Ｃｍｅｘでは式の評価に遅延評価を用いている．遅延評価では，結果を得るために必要な部分式のみを評価する．すなわち，式はその値が必要とされるまで評価が遅延される．この遅延評価を用いると，高階関数や無限リストを簡単に取り扱うことができるため Miranda や LML[7] 等いくつかの関数型言語でも採用されている．しかし並列処理を行う場合，遅延評価では次の問題点が考えられる．

(1) プロセス間通信の場合，転送するデータが未評価の式のままであると通信にかかるコストが非常に高くなる．
(2) 評価に先だってプロセスを生成しておくことができない．

したがって，このような場合は一部の部分式の先行評価を行う方が望ましい．

Ｃｍｅｘでは preset 句を用いて任意の部分式の先行評価を行うことができる．preset 句は where 句と同様に用いる(図12)．

```
func args =  ((.. a) .. b)..)
                preset
                  a = preset-expression-1
                  b = preset-expression-2
```

図12　preset句

ユーザは preset 句を用いて，関数の定義内で局所的な式の定義を行う．preset 句内の式は，identifier = expression の形をとり，関数の定義などは許されない．そして preset 句内のすべての式は，関数の式の評価に先だって上から順に評価される．
図13 に preset 式の文法を示す．

prset-clause ::= preset *pr-state*$^+$
pr-state ::= *ident* = [{p}] *expr*
| useport {*ident* [:: *type*]}$^+$
| array *ident* (*ident*, 0, *expr*) = *expr*

図13　preset 句の文法

3.2　プロセス

プロセスは並列処理の単位であり評価の過程で動的に生成される．プロセスは preset 句内においてプロセス生成子 {p} によって生成される（図14）．

```
{p} :: * -> *                    || type
Identifier = {p} Expression      || description
```

図14　プロセスの生成

生成されたプロセスは，与えられた Expression の評価を行い，正規形が得られたならば Identifier に値を返し終了する．結果が返って来る前に Identifier の値を評価しようとしたプロセスは値が返るまで封鎖される．

3.3　ポートとプロセス間通信

ポートはプロセス間の通信の仲介を行う．送信側のプロセスはポートに値を出力，受信側のプロセスは同じポートから値を取り出すことにより通信が行われる．この場合のデータの送受信は同期方式をとっている．すなわち送信側と受信側が共にそろったとき初めてデータの受け渡しが行われる．どちらか一方がそろわないなら，もう一方は封鎖される．

ポートは宣言子 useport によって宣言され，評価の過程で動的に生成される．このポートの型を表すため新しくポート型を基本の型に追加した．ポート型は port * で表され，* はそのポートに対して入出力される値の型を示す（図15）．

```
num_port :: port  num
tpl_port  :: port  (char, num)
```

図15　ポート型

図15 で，num_port は整数の入出力を行うポート，tpl_port は文字と整数のタプルの入出力を行うポートであることを示している．

ポートは受け取った値をリストの形で保持する．ポートに値を出力することはそのリストの後ろに値を追加することであり，ポートから値を取り出すことはポートの受け取った値のリストを得ることである．したがって，ポートから値を受けとるプロセスは，ポートからの出力をリストと全く同じものとして取り扱うことができ，プロセス間通信を意識しなくてもよい．

ポートへの入出力操作を行うために二つの命令が用意されている（図16）．

```
|| ポートに値を出力する
{>} :: port * -> * -> *        || type
{> Portname} Value             || description

|| ポートの値のリストを得る
{<} :: port * -> [*]           || type
{< Portname}                   || description
```

図16　ポートの入出力命令

{>} は Value を指定されたポートに出力し Value をそのまま返す．{<} は指定されたポートの値のリストを返す．

ポートは，他の関数に引数として渡すことができる．したがって，一つのポートを複数のプロセスが共有することによって，多対多の通信を行うことができる．多対多の通信では，複数のプロセスが同じポートに値を出力した場合，ポートは先着順に値を保持し，また複数のプロセスが同じポートから値を得ようとした場合，すべてのプロセスはそのポートの値のリストを共有する．

3.4　array 文

array 文を用いて，同じような要素の集合を中身とするリストを生成することができる．array 文は，識別子と (変数，初期値，終値) からなるタプルをとる．そして，タプルの中の変数を初期値から終値まで変化させ，定義式に基づいてリストの各要素を生成する．リストの要素の生成は先行評価によって行われる．並列処理において同じ様な処理を行うプロセスを複数個生成しなければならないことが多いが，Ｃｍｅｘでは array 文を用いることにより簡単に行うことができる．図17 に array 文を用いたフィボナッチ数を求める関数の定義を示す．

```
fib n = f!n
        preset
         array f (m,0,n)
          = {p} 1                  , m < 2
          = {p} f!(m-1)+f!(m-2), otherwise
```

図17　array 文を用いた関数の定義

4.　実現

4.1　コンパイラ

Ｃｍｅｘのソースプログラムは，まずタイプチェッカ[8] によって，型の使用に関して矛盾がないかどうか調べられたのち，パターンマッチングコンパイラ[9] によって通常のλ式に変換される．そして，λ式は G-マシン[10] のコードに変換される．G-マシンは，グラフリダクションを行う仮想的なスタックマシンであり，そのコードは，スタックに対する PUSH などターゲットマシンのコードに対応したものになっている．

Ｃｍｅｘでは，独自の機能の実現の為に以下の命令をＧコードに追加している．

TPL n
　スタック上の n 個をタプルにする

CONSTRUCT s, n

構成子 s に番号 n を与える

PUSHCON s

構成子 s を作成

GETCON

構成子番号をVスタックにプッシュ

PARALLEL

プロセスの生成

PORTIN

ポートからの入力

PORTOUT

ポートへの値の出力

4.2 評価部

評価部は，現在，G-コードを読み込んでG-マシンのシミュレートを行う仮想G-マシンであり，SUN3上で実装されている．並列機構は SunOS 4 の LWP (LightWeight Processes)[11] を用い，評価部を複数作り出すことによってプロセスを実現している．また、プロセス間通信の低レベルルーチンにはLWP のライブラリを用いている．

現在，分散オペレーティングシステムの開発[12] が同時に行われており，その分散オペレーティングシステム上での実装を行う予定である．

5. 応用例

5.1 セマフォ

セマフォとは非負の整数値を持つ変数であり，プロセスの排他制御を行うために用いられる．

セマフォの値に対する操作は，P 操作と V 操作の二つがある．

P(S)： S > 0 ならば S の値を一つ減らす．
S = 0 ならば待行列に入る．

V(S)： Sの値を一つ増やす．

Ｃｍｅｘではセマフォをポートと二つのプロセスで実現する．セマフォの整数値はポートのリストの長さで表される．そして，P 操作によってリストの要素数を一つ減らし，V 操作によって要素数を一つ増やす．P 操作の際，ポートのリストに要素がない場合，P 操作を行ったプロセスは値を受けとれないために封鎖される．封鎖されたプロセスは，ポートに値を送るつまり，V 操作によって起こされる．

図18にセマフォの生成を行う関数 semaphore の定義を示す．実際の P，V 操作は関数 semaphore の返値の p ポート，v ポートに値を出力することによって行う．

```
semaphore n
 = (p,v)
   preset
    useport p, v
    pp = {p}p_proc ([1..n]++{<v}) {<p}
    vp = {p} v_proc {<v}

p_proc :: [*] -> [**] -> ***
p_proc (n:ln) (m:lm) = p_proc ln lm

v_proc :: [*] -> **
v_proc (m:lm) = v_proc lm
```

図18　セマフォの記述

5.2 食卓の哲学者の問題[13)]

複数のプロセスを用いた応用例として食卓の哲学者の問題を考える．食卓の哲学者の問題とは次の通りである．

「五人の哲学者が丸いテーブルに座っており，それぞれ思索をしたり食事をしたりしている．テーブルの中心にはスパゲティが置いてあり，哲学者は自分の右と左にあるフォークを用いてそれを食べる．フォークは哲学者の間に一本ずつしかおいてないため，一人の哲学者が食事中のときはその右と左に座っている哲学者は食事をすることができない．このような状況でどの哲学者も飢死にする（デッドロック等によって食事ができない）ことのないようにする．」

プログラムを 図19 に示す．哲学者は最初に右側のフォークを取る．しかし全員が同じように右側のフォークを取った場合，誰も左側のフォークを取ることができずデッドロックに陥る．そこで，同時に食事ができるのは4人までとする．そうすることにより一人は必ず両方のフォークを取ることができ，デッドロックは起こらない．これを table を用いて行っている．table は値が 4 のセマフォであり，各々の哲学者は食事に入る前に table に対して P 操作を行い，食事が終ったのち V 操作を行う．また，フォークは値が 1 のセマフォ fork で実現している．五本のフォークと五人の哲学者の為のプロセスを生成するために array 文を用いている．

```
main =[]
        preset
          table = semaphore 4                                  || tableのセマフォの生成
          array fork (n,0,4) = semaphore 1                     || フォークのセマフォの生成
          array man (n,0,4)                                    || 哲学者プロセスの生成
            = {p} phils table (fork!n) (fork!((n+1)mod5))

phils (tp,tv) (rp,rv) (lp,lv)
    = phils (tp,tv) (rp,rv) (lp,lv)                            || preset以下の繰り返し
      preset
        think       = thinking                                 || 思索
        start_eat   = {> tp} 1                                 || 食事を始める
        take_right  = {> rp} 1                                 || 右のフォークを取る
        take_left   = {> lp} 1                                 || 左のフォークを取る
        eat         = eating                                   || 食事
        put_left    = {> lv} 1                                 || 左のフォークを置く
        put_right   = {> rv} 1                                 || 右のフォークを置く
        end_eat     = {> tv} 1                                 || 食事を終わる
```

図19　食卓の哲学者の問題の記述

6. おわりに

分散型関数型言語Ｃｍｅｘの開発について述べた．Ｃｍｅｘは，プロセスとポートを使ったプロセス間通信を用いて並列処理を可能にした．また，preset 句を用いた部分式の先行評価を行い，並列処理における遅延評価の問題点を解決している．

本文では，Ｃｍｅｘの基本機構について述べた後，Ｃｍｅｘを用いて実際の並列処理のプログラミング例を示し，その応用性確かめた．

Ｃｍｅｘは現在処理系を構築中であり，処理系の完成後，実際の評価を行う予定である．

参考文献

1) M. C. J. D. van Eekelen, E. G. J. M. H. Nocker, M. J. Plasmeijer and J. E. W. Smetsers, Concurrent Clean, Technical Report 89-18, University of Nijmegen
2) M. C. J. D. van Eekelen, M. J. Plasmeijer and J. E .W. Smetsers, Communicating Functional Processes, Technical Report 89-3, University of Nijmegen
3) M. C. J. D. van Eekelen, M. J. Plasmeijer, J. E. W. Smetsers, Parallel Graph Rewriting on Loosely Coupled Machine Architectures, Techical Report 88-9, University of Nijmegen
4) D. A. Turner, Miranda : A Non-Strict Functional Language with Polymorphic Types, International Conference on Functional Programming Language and Computer Architecture, LNCS, Vol.201, pp.1-16, Springer-Verlag, 1985
5) P. J. Landin, The Next 700 Programming Languages, Commun. ACM 9, pp.157-166, 1966
6) R. Milner, ATheory of Type Polymorphism in Programming, Journal of Computer and System Sciences, Vol.17, pp.348-375, 1978
7) L. Augustsson, A Compiler for lazy ML, Proceedings of the ACM Symposium on Lisp and Functional Programming, pp.218-227, 1984
8) J. Fairbairn, A New Type-Checker for a Functional Language, Science of Computer Programming, Vol.6, pp.273- 290, 1986
9) P. Wadler, Efficient Compilation of Pattern-Matching, Chap. 5, The Implementation of Functional Programming Languages, Prentice / Hall International, 1987
10) T. Johnsson, Efficient Compilation of Lazy Evaluation, Proceedings of the ACM Conference on Compiler Construction, pp.58-69, 1984
11) System Services Overview, Sun Microsystems Inc.
12) 油谷聡，藤原正之，白川洋充，大野豊，分散オペレーティングシステムＴｈｅｔａの構成，情報処理学会第41回全国大会，pp.4-117,4-118
13) E. W. Dijkstra, Co-operating Sequential Processes, in Programming Language, Academic Press, 1968