2E−5 関数型言語 Valid への並列オブジェクト指向プログラミング機能の導入

日下部茂　友清孝志　谷口倫一郎　雨宮真人

九州大学総合理工学研究科

1. まえがき

知識処理など高度の情報処理システムとして、並列協調型処理システムが有望視されている。並列協調処理システムとは、多数の処理体が互いにメッセージをかわして自律的に動作し、協調して問題を解くようなシステムのことである。

メッセージ交換に基づくモデルとして actor モデルが提案されており[1]、またさまざまな並列オブジェクト指向プログラミングが提案されている[2]。我々は前述のようなメッセージフローシステムを実現するには、データフローの概念を基礎におく計算機方式が有効であると考え Datarol アーキテクチャを提案している[3]。データフロー・メカニズムにより処理体間の並列性だけでなく、処理体内の並列性も利用することができる[4]。

我々は、現在データフロー用関数型言語 Valid[5]を用いてプログラムを書き、それを Datarol プロセッサ用のコードにコンパイルしている。本論文では、並列協調型の問題の解決法を記述するため、我々が用いている関数型言語 Valid に並列オブジェクト指向の概念を導入することについて考察する。

2. 並列オブジェクト指向

並列協調型システムは多数の処理体（プロセス）により構成される。エージェントは動的に生成されまた動的に消滅する。各プロセスは、消滅するまで活性化状態にあり一個の独立した計算主体として自律的に動作する。各プロセスはメッセージによって交信を行い並列に動作する。

このようなメッセージ・フローによる並列協調型の問題を自然に記述するため、並列オブジェクト指向の概念をデータフロー用関数型言語 Valid に取り入れる。

3. 基本概念

以下に並列協調型のシステムを記述するのに必要と思われる基本的な概念について考察する。

3.1 モジュール

並列に動作する処理体の基本単位としてモジュールを考える。このような処理体は一般にオブジェクトと呼ばれることが多いが、我々が提案しているシステムの処理体ではデータフロー概念に基づいた処理を行うためこのように呼ぶことにする。モジュールは局所状態、処理ルーチンを持つ。処理ルーチンはパターン照合によって選択的に起動され、メッセージに対応した局所状態の更新を行う。

またモジュールはメッセージ・バッファを持ち、そのメッセージ・バッファには名前が与られる。その名前が参照されるときに先頭のメッセージが取り出される。メッセージの受信とモジュールの本体の処理は非同期に行われる。　以下にモジュール記述の基本形を示す

```
module モジュール名 (局所状態の初期値)
    = { let 定義 ···
                ···
        recurse with (次メッセージ, 新しい状態値)
          where  本体定義
                ··· }
```

高い並列性を取り出せるよう、実行時に動的にインスタンスの生成、消去ができるようにする。

・インスタンスの生成

P=new(M,Init-states)

new によりモジュール M の新しいインスタンス P が初期状態 Init-states で生成される。

・インスタンスの消去

s=erase(module-instance)

erase により module-instance に自分を消滅するよう指令するメッセージが伝えらえる。module-instance がそのメッセージを処理すると module-instance は消去され、その終了シグナルが s に送られる。シグナル s の確認が必要ない時は次のように記述する。

!=erase(module-instance)

ここで!=は返される結果を捨て去ることを表す。

3.2 メッセージ

メッセージ交信は次のように記述する。

s=send(module-instance,[message|destination])

これは module-instance に対し message を送出し、その message に対する最終的な処理結果が求まればその結果を指定された返答先の destination に送ることを表す。destination ではモジュールインスタンス名とそのインスタンス内の場所を指定する。結果をどこにも返す必要が無い場合、destination は省略される。s には send の実行完了シグナルが返される。実行を確認する必要が無い場合、左辺は!と書く。

返答先がメッセージを発するモジュール自身のインスタンス内の場所 x である場合は次のように記述することも可能とする。

(s,x)=send(module-instance,message)

Introduction of Parallel Object-Oriented Programming Feature to Functional Programming Language Valid

Shigeru KUSAKABE, Takashi TOMOKIYO, Rin-ichiro TANIGUCHI and Makoto AMAMIYA

Graduate School of Engineering Sciences, Kyushu University

この場合 x の値に依存する処理は x の値が求められるまで待たされるが、依存しない部分はデータフローメカニズムに従い処理を進めることができる。

結果を求めるために、モジュール間で処理をたらい回しにする必要がなく、1ステップのメッセージ交信で十分な場合には次のように書くことを可能とする (Call型)。

x=call(module,message)

call型のメッセージの発信元に返答を送る場合次のように記述する。

sまたは!=reply(message)

同期を明示的に表したい場合には次のように記述する

s=send(···),
exp after s

これは確認シグナルがsに返って来るまでexpの処理を待つことを表す。

4. 実現機構

以上のような処理機構をデータフロー概念に基づいて実現する[4]。new M によってモジュール M の新しいインスタンス P_0が生成される。P_0に送られたメッセージはバッファの中に一旦取り込まれる。メッセージ中のパターンとのパターン照合によって処理ルーチンが起動される。処理ボディのインスタンスは動的に生成され、状態の更新がある場合新しい状態値はフィードバックされる。依存の無いメッセージは並列に処理される。このような機構はデータ駆動の原理によって行われる。

5. 例題

本言語によるプログラム記述の例として、覆面算の求解プログラムを記述した。覆面算とは、例えば次のような式において、各文字に対応する数を決定する問題である。

```
    D O N A L D
+   G E R A L D
-------------------
  R O B E R T
```

データ依存関係のみを記述していくことで、プログラムは構成される (図1)。また、1つのインスタンスに送られた複数のメッセージのうち互いに依存関係の無いものは、データ駆動により、自動的に並列実行される。さらに Transfer 型送信を用いることにより、複数の問題をパイプライン的に並列実行していくことが可能となる。

6. まとめ

本論文では並列協調型の問題解決を記述するために必要な、並列オブジェクト指向の概念を関数型言語 Valid に導入することについて考察した。

参考文献

[1] Gul A. Agha : "Actors: A Model of Concurrent Computation in Distributed Systems", MIT Press(1986).

[2] A. Yonezawa and T. Mario : "Object-Oriented Concurrent Programming", MIT Press(1987).

[3] 雨宮 : "超並列多重処理のためのプロセッサ・アーキテクチャ","コンピュータアーキテクチャ」シンポジウム論文集, pp.99-108(1988).

[4] 雨宮, 長谷川:"並列協調システムにおけるメッセージ処理機構", ソフトウェア科学会第4回大会論文集, pp.315-318(1987).

[5] 長谷川, 雨宮 : "データフローマシーン用関数型高級言語 Valid", 電子情報通信学会論文誌 D, Vol.j71-D, No.8, pp.1532-1539(1988).


```
Print:module(device)
= -- メッセージとして受け取った結果を印字する.

Fukumen:module(dest)
= { let found:function(collist)
        = -- その桁の確定文字の個数を返す
  in for (mess):(message_queue()) do
    recurse with next(mess)
  where
    [alist,free_num,problem,carry]=mess,
    [mycol|leftcol]=problem,
    x=first(mycol), xnum=cdr(assoc(x, alist)),
    y=second(mycol), ynum=cdr(assoc(y, alist)),
    z=third(mycol), znum=cdr(assoc(z, alist)),
    !=case{
      found(mycol)=3
      → {let res = xnum + ynum + carry
          in if leftcol=[ ] and res=z then
               send(dest,alist)
             elsif res=z then

send(dest,'[alist,free_num,leftcol,0])
             elsif mod(res,10)=z then

send(dest,'[alist,free_num,leftcol,1])
             else ''failed'' },
      found(mycol)=2
      → case{
          null(xnum) →
          {let k=znum − ynum − carry,
              xnum=if k ≥0 then k else k+10
          in if member(xnum, free_num) then
              send(self,[[[x|xnum]|alist],remove(
xnum, free_num),problem,carry) ]
             else ''failed'' },
          null(ynum) → ··· ,
          null(znum) → ··· },
      found(mycol)=1
      → case{
          null(xnum)) →
            for each i in free_num do
              send(self,[[[x|i]|alist],remove(
i, free_num),problem,carry]),
          null(ynum) → ··· },
      found(mycol)=0
      → for each i in free_num do
          send(self,[[[x|i]|alist],remove(i,
 free_num),problem,carry]) }}

-- 起動: --
{let
print=new(Print(TTY1)),
col6=new(Fukumen(print)),
col5=new(Fukumen(col6)),
...
col1=new(Fukumen(col2)),
digits='[0 1 2 3 4 5 6 7 8 9],
problem='[['D','D','T']['L','L','R']['A','A','E']
        ['N','R','B']['O','E','O']['D','G','R']],
ss=for each i in digits do
    send(col1,[[caar(problem)|i],
                remove(i,digits),problem,0]),
!=erase(col6) after ss,
...
!=erase(col1) after ss,
!=erase(print) after ss  in ss}
```

図 1:覆面算のプログラム例