3E−3

ストリームベースのオブジェクト指向言語におけるメッセージリダクションの並列化

吉田 実, 田中英彦
{minoru,tanaka}@mtl.t.u-tokyo.ac.jp
東京大学工学部*

1 はじめに

Comitted Choice Language(CCL) をベースにしたオブジェクト指向言語はCCLの柔軟なプログラミングを可能にする特徴を受け継ぎながら、ユーザからは副作用としても見えるインスタンス変数をもつなど、よりプログラミングし易い言語を目指している。本論文はメッセージの具体化の程度により可能な限り先まで計算をするめることに関して考察する。その場合に当然のことながらインスタンス変数のアクセスパタンに大きな影響を受けることになる。

2 ストリームベースのオブジェクト指向言語

ストリームをベースにしたオブジェクト指向言語は、Flengで書けば、

```
object([Message|Messages],InstanceVariable) :-
    listener(Message,Messages,InstanceVariable).
listner(message(XX),Messages,InstanceVariable) :-
    (メッセージの実行部分),
    object(Messages,InstanceVarible).
```

のようにインプリメントされているか、もしくは、そのように解釈できる。しかし、このようなインプリメントは本来実行できる筈の処理を行なえず、その結果、並列性をそこなうことがある。本論文では、そのような不要な処理の停止が起きる場合を指摘し、その停止ををなくし、最大の並列性を引き出す手法を提案する。

3 メッセージリダクションのサスペンドによるメッセージ受信の停止

述語リスナのあるクローズのヘッドの第一引数が、message(a)であった場合、message(X) と言うメッセージを受信した場合、そのオブジェクトのメッセージの解釈の実行は更にメッセージが来ている場合でも、そこで停止してしまう。

しかし、上の解釈実行プログラムを次のように変更することでメッセージの受信を停止しないようにできる。

```
listener(message(X),Messages,InstanceVariable) :-
    message(X,Messages,InstanceVariable,
        NewInstanceVariable),
    object(Messages,NewInstanceVariable).
message(X,Message,InstanceVariable,
    NewInstanceVariable) :-
    (メッセージの実行部).
```

そのメッセージがインスタンス変数の書き込みをしていないことがわかっていれば、次のように最適化できる。

```
listener(message(X),Messages,InstanceVariable) :-
    message(X,Messages,InstanceVariable),
    object(Messages,InstanceVariable).
message(X,Message,InstanceVariable) :-
    (メッセージの実行部).
```

処理系が変数のメッセージの送信を許している場合は、上のプログラムでは、まだ、リスナリダクションを停止させてしまう。

```
object([Message|Messages],InstanceVariable) :-
    listener(Message,InstanceVariable,
      NewInstanceVariable).
listner(message(X),InstanceVariable,
  NewInstanceVariable) :-
    message(X,InstanceVariable,
      NewInstanceVariable).
message(X,Message,InstanceVariable,
  NewInstanceVariable) :-
    (メッセージの実行部).
```

のようにすれば、メッセージの受信は停止しない。

4 メッセージの逐次関係指定によるメッセージ受信の停止

ストリームに基づくオブジェクト指向言語において、なぜストリームを使うかというと自然に順序関係が指定できるからである。ストリームを複数のオブジェクトが共有する場合には、マージ(並列)関係とアペンド(逐次)関係がある。マージ関係の場合は、各ストリームをマージを介してオブジェクトを表すストリームへつなぐ。この結果、各ストリームから非同期にメッセージを送出できる。一方、アペンド関係の場合は、同様にアペンドを介することになる。Stream1,Stream2がアペンド関係にある場合、共有変数の関係は、

```
append(Stream1,Stream2,Messages),
object(Messages,InstanceVariable)
```

のように表される。この場合、Stream2はStream1がターミネートするまでobjectに渡されることはない。しかし、この場合、Stream2を先に実行しても構わない場合がある。

```
append(Stream1,
  [readInstanceVariable(LastInstanceVariable)],
  Messages),
object(Messages,InstanceVariable),
```

*"Parallel Reduction of Stream-based Object-oriented Languages", YOSHIDA Minoru and TANAKA Hidehiko, Univ. of Tokyo, Faculty of Engeneering

```
object(Stream2,LastInstanceVariable)
```

上のようにすることで世代のオブジェクトが同時に走ることができ、並列性をあげることができる。

5 本手法適用の限界

本手法の問題点となると思われる点について挙げ、実は問題点ではないということを明らかにする。

5.1 I/O 処理を直接行なう述語呼びだし

I/O 処理を直接行なう述語呼び出しをストリームの逐次制御を用いて記述したい場合があるかも知れない。本手法がそのことを元のインプリメントと同程度に行なえることを示す。メッセージの来た順番に書き込んで行くメソッドの定義は以下のようになる。

```
:write(X) :-
    write(X,Res), :wait(Res).
:wait(true).
```

これは、write/2 が終るまで、:wait(Res) というメッセージでサスペンドさせておくわけである。本手法では、メッセージのリダクションで停止するようなことはないのでこのように順序制御にもちいることはできない。

本手法では、インスタンス変数を用いて述語が順番に動くように制御する。

```
:write(X) :-
    :wait :=  Wait,
    wwrite(*wait,Wait).
wwrite(true,Wait) :-
    write(X,Wait).
```

次の世代のオブジェクトに対して、write/2 の終了をインスタンス変数を用いて知らせている訳である。前の処理の終了をメッセージを使わずにインスタンス変数を用いて知らせていると言える。

マクロを使えば、

```
:write(X) :-
    *wait := Wait,
    (wait(*wait) -> write(X,Wait)).
```

とも書ける。

5.2 インスタンス変数に未定義変数を代入できる場合

インスタンス変数に未定義変数を代入できる場合は、インスタンス変数に代入したかどうかを示すことが必要になる場合がある。これは、例えば、以下のように変数を代入すれば良い。

```
listener(write(X),Messages,InstanceVariable) :-
    object(Messages,iv(X)).
```

上の例では、InstanceVariable が未定義なら、まだ書き込みが行なわれてないことを、iv(X) の形なら書き込まれたことを示す。しかし、この書き込まれたかどうかの判別はインスタンス変数が変数ならある処理をするという場合にしか必要ない。そのような書き方は通常の CCL の作法から外れるので使用頻度は低いと考えられる。

6 並列解釈について

ストリームを使って送られてきたメッセージを順番に読んで解釈するのは、それなりに合理的なものであるが、メッセージが集中しやすいオブジェクトでボトルネックとなるおそれがある。メッセージが並列解釈できれば、オブジェクト内部の並列分散処理を更にすすめることができる。そのためには、ストリームベースの順序制御をあきらめ、メソッドが呼ばれた時にアトミックに行なうことは必要なインスタンス変数の読み書きを処理するだけという方式が考えられる。現在、実験的インプリメントをしている Fleng++ では、この方式を採用している。この方式では、オブジェクトはストリームに流れるメッセージを逐次に解釈するプロセスではなく、副作用のあるアトミックにアクセスできる構造体のように見える。この場合、デフォルトではマージ関係になるが、アペンド関係の制御は可能である。先に実行して欲しいメソッドにオブジェクトの入るべき変数を渡す。これには、必要なインスタンス変数の領域が確保されている。次に実行して欲しいメソッドはそのオブジェクトへメッセージパッシングする。先に実行したメソッドは次のメソッドの実行を行なっても良くなったら、オブジェクトの入る変数のインスタンス変数をバインドする。

7 インスタンス変数の単一参照性について

インスタンス変数にシステムの管理テーブル、オブジェクトの集合などの比較的大きなベクタを代入したい場合がある。そのような巨大なベクタを入れている場合には、その単一参照性を維持しなけば、破壊代入ができなくなる。その結果、ベクタの一要素を変更したい場合に他の全要素をコピーしなければならなくなる。これは通常許されないオーバヘッドとなる。

単一参照性を維持するレベルは、1) 深さが 1 のベクタの構造だけの場合と 2) 構造データの全てに渡る場合というのがプログラミング上で必要なもののほとんどである。1) の場合は初期値を代入する時に単一参照データでなければベクタの 1 段分の構造だけコピーする。その後はベクタの読み出しはコピーして渡し、書き込みはそのまま破壊代入する。この場合の処理は 1 セル分のコピーや書き込みなので非常に軽い。2) の場合も同様であるが、コピーは場合によっては重い処理になる。Fleng++ では、単一参照性を保持する範囲はプログラマによって指定される。

8 おわりに

ストリームベースのオブジェクト指向言語では、処理をなるべく速くすすめるためには、注意深いインプリメントと若干のオーバヘッドが必要になる。一方、逐次解釈の原則をなくしてしまえば、オブジェクトを副作用を持つインスタンス変数を保持する構造として捉えられ、並列分散処理を行なえる範囲が広がる。

なお、本研究は文部省特別推進研究 No.62065002 の一環として行なわれているものである。

参考文献

[1] 吉田実, 舘村純一, 勝亦章善, 田中英彦, “Fleng++ 実験環境”, 第 39 回情報処理学会全国大会 (1990)