

FGHC による
オブジェクト指向プログラミングのための
トランスレータ

小西弘一

東京大学工学部計数工学科

FGHC 等の並列論理型言語の重要なプログラミングスタイルとして、オブジェクト指向スタイルがある。しかし、このスタイルで書かれたプログラムは必然的に、冗長な記述や意図の読み取りにくい記述を多数含んでいる。そこで、簡潔で明瞭な記述のために、FGHC プログラムへの展開を前提とする上位言語が必要である。ここでは、FGHC による一般的なオブジェクト指向スタイルのプログラムの記述効率は、比較的単純なマクロトランスレータにより大きく改善できることを示す。

A TRANSLATOR FOR THE OBJECT-ORIENTED
PROGRAMMING IN FGHC

Koichi KONISHI

Department of mathematical engineering and information physics
faculty of engineering, university of Tokyo
7-3-1 Bunkyo-ku, Tokyo, 153, Japan

The object-oriented style is one of the most important programming styles for the parallel logic programming language like FGHC. The program written in this style, however, inevitably has many redundant and obscure descriptions. The author made a translator which provides a form to eliminate the description of the untouched internal state of the object. It also provides some macros to handle the stream neatly. This paper describes the specification of the translator, and shows how it enables the terse and clear description of the program of the object-oriented style.

1 FGHC によるオブジェクト指向プログラミング

1.1 オブジェクトの構成

並列論理型言語では、一般に、述語を末尾再帰的に呼び出し続けるプロセスによって一つのオブジェクトを表す。

オブジェクトには普通、ストリーム通信の変数を持たせ、このストリームを通じてメッセージを送る。オブジェクトはこのメッセージを、オブジェクトが行なう処理を選択するセクタとして扱う。オブジェクトはセクタを受け取るたびにセクタに応じた処理、たとえばインスタンス変数の値の更新や、別のゴールの呼びだし、あるいは他のストリームを通じての通信などを行い、そして再帰呼び出しを行う。次のメッセージは、再帰呼び出しによって起動されたゴールが処理することになる。

このオブジェクトのインスタンス変数は、そのゴールの引数にその値を保持し、再帰呼び出しのたびに新しいゴールにその値を渡すことによって表す。インスタンス変数の値の更新は、再帰呼び出しの際に新しいゴールに渡す値を変えることによって行う。

FGHC によるオブジェクトの最も簡単な例としてしばしば提示されるものに、以下に示すようなカウンタがある。

このカウンタを表すオブジェクトは、カウンタの値を示すインスタンス変数を持つ、外部から送られるセクタに応じてインスタンス変数の値の増減などの処理を行う。例えば、つぎに示すクローズはこのオブジェクトの定義の一部で、セクタ `up` を受け取った時にカウンタの値を 1 増やすことを定義している。

```
counter([up|Next],Value):-
    true|
    New_value:=Value+1,
    counter(Next,New_value).
```

述語 `counter` の第一引数はセクタを受け取るためのストリームを保持している。また、第二引数 `Value` はカウンタの値を示すインスタンス変数である。ここでは、クローズの頭部で、送られてきたセクタが `up` であることを調べている。実際に `up` を受け取った場合の処理はボディ部に記述されている。すなわち、まず `Value` に

1 を加えた値を計算し、これを第二引数に与えて `counter` を再帰的に呼び出している。

1.2 自分自身への送信

オブジェクトが自分自身にメッセージを送るには、メッセージ受信用ストリームに送ろうとするメッセージを加えて、再帰的に呼び出す次のゴールに渡せばよい。たとえば、`up_twice` というセクタを受け取った時、セクタ `up` を二つ、自分自身に送るには、次のように記述すればよい。

```
counter([up_twice|Next],Value):-
    true|
    counter([up,up|Next],Value).
```

自分自身へのセクタの送付はローカルな述語の呼び出しであるということができる。何かがローカルであるというときは、それが場所に依存するということを述べている場合と、その存在が公開されていない、他からはアクセスできないということを述べている場合がある。あるオブジェクトの中で定義されたセクタは、別のオブジェクトに対しては、別の意味を持つことができる。したがって、前者の意味で、セクタはそのオブジェクトにローカルである。

また、オブジェクトのインスタンス変数は、同じクラスに定義されているすべてのローカルな述語から共通にアクセスできるという意味で、グローバルな変数である。

1.3 オブジェクト指向スタイルの冗長性

インスタンス変数の値を頭部で引数として受け取り、これを再帰的に呼び出すゴールに渡す、という処理はほとんどのクローズに共通する処理である。次のゴールに渡す値は、メッセージに対応する処理の内容によって、あるものは現在のそのままの値、また更新するものについては新しい値でなければならない。

10 個のインスタンス変数を持つオブジェクトの場合、ある一つのインスタンス変数の値を更新するクローズを記述しようとする、図 1 のようになる。再帰的に呼び出すゴールには、目的のインスタンス変数の新しい値だけではなく、変更を加えなかった残りの 9 個の変数の値も引数と

```
obj([set_g(X)|Next],A,B,C,D,E,F,G,H,I,J):-
    true|
    obj(Next,A,B,C,D,E,F,X,H,I,J).
```

図 1: 10 個の引数を持つオブジェクト

して与えなければならない。つまり、10 個のインスタンス変数を持つオブジェクトを定義するクローズには、そのクローズが更新しようとする変数の個数がいくつであろうとも、必ず 10 個の引数を記述しなければならない。

このような記述を、そのオブジェクトが受け取ることができる各メッセージに対応するクローズについて繰り返すのは冗長であり、再帰的に呼び出すゴールに渡す変数の個数や順序などに間違いが生じる可能性が高くなる。さらに、このプログラムを読もうとする人にとっても、たとえば上記のクローズで、どの変数の値が更新されたかを一見して認識することは困難であり、プログラムの意図が明瞭に現れていない。

また、ストリームを通じた通信の記述にも、冗長で、プログラムの意図が伝わりにくい表記を使用しなければならない。

このような問題は、トランスレータによって解決できる。簡潔な構文を定め、その構文をトランスレータで FGHC によるオブジェクト指向スタイルに変換すればよい。

以下では、以上に述べた目的で筆者が試作したトランスレータについて、まず現在の機能を説明し、次に問題点を挙げ、改善案について考察する。

なお、このトランスレータは、暫定的に Maglo と命名されている。

2 Maglo の構文と機能

2.1 クローズの頭部の省略

一つのクラスを定義するクローズは、原則として、どれも同じ述語を頭部に持っている。したがって、その述語をクラス定義の先頭で指定して、各クローズでは頭部を省略するのが望ましい。現在は図 2 のような構文を使用している。すなわち、class...endclass. によって範囲を定め、class の直後で述語の雛型を記述している。

もちろん、同じ述語を頭部に持つ、といっても、同じなのは functor 名と arity だけで、一般に各引数はクローズによって異なっている。しかし、定数項を引数として持つ頭部は、変数を引数に持つ頭部と、その変数と先の定数項の同一化を行なうガード部の組合せで置き換えることができる。そこで、省略する頭部はすべての引数が変数であるような述語であるとし、この述語の引数であるべき定数項は、ガード部に、対応する変数との同一化を行うゴールとして記述する。たとえば、FGHC で次のように記述するクローズは、

```
foo(sugar,Cup,Beverage):-...
foo(milk,Cup,tea):-...
foo(pour,demitasse,Beverage):-...
```

class...endclass. で囲んだ領域内では、次のように記述する。

```
class foo(Action,Cup,Beverage);
Action=sugar->...;
Action=milk->Beverage=tea|...;
Action=pour->Cup=demitasse|...;
endclass.
```

-> は、以下で説明するが、通常の FGHC の演算子:- に相当するものである。なお、この例では、ガード部のゴールの一つを、省略した述語のかわりに頭部に置いている。

2.2 再帰呼び出しを行うゴールの省略

再帰的に呼び出されるゴールも、頭部と同じ functor 名と arity を持つ構造体で記述される。オブジェクト指向スタイルではほとんどのクローズが再帰呼び出しを行うから、特に指定しない場合には再帰呼び出しを行うものとして、class...endclass. で囲んだ範囲内では省略できるようにする。ただし、再帰呼び出しを行わないクローズも一応記述できるようにしておく。再帰呼び出しを行うかどうかは、クローズの頭部と本体を結ぶ演算子(通常の GIIC では:-

```
class <functor 名>(<インスタンス変数名のリスト>);
<メソッドを定義するクローズのリスト>
endclass.
```

図 2: クラス定義の構文

で表されるもの)に次のようなものを使用することにより指定する。

- <頭部>-><本体>;
再帰呼び出しを加える。
- <頭部>-:<本体>;
再帰呼び出しを加えない。

2.3 インスタンス変数の値の更新の記述

FGHC によるオブジェクトのインスタンス変数の値の更新は、再帰的に呼び出すゴールに新しい値を渡すことにより行う。しかし、再帰的に呼び出すゴールの記述は省略することにしたので、値の更新を別に示す必要がある。これは次のようなマクロを疑似ゴールとしてボディ部に置いて指定する。

<変数名>/<新しい値>

新しい値として算術式を与えると、その式を評価した値を使って更新を行う。

更新した値は、そのクローズの中では、そのインスタンス変数の名前では参照できない。その名前を使ってそのクローズの中で参照できるのは、クローズが頭部で受け取った元の値である。更新した値が参照できるようになるのは、再帰的に呼び出すゴールの中からである。同じ変数を更新するマクロが一つのクローズに複数あると、最後のものだけが効力を持つ。ここで、以下で述べる通信用マクロも値の更新を行うことに注意する必要がある。

2.4 通信の記述

FGHC でメッセージの送受信を記述するときには、

- 今回の通信に使用する共有変数
- メッセージ

- 次回の通信に使用する共有変数

の三つを指定する。しかし、次回の通信に使用する共有変数は、通常は同じクローズの中では（再帰的に呼び出すゴールに渡す時以外には）参照しないので、省略できると便利である。

そこで、通信に使用する変数がインスタンス変数の場合には、図 3 のような疑似ゴールによって通信を記述し、次回用変数の指定を省略する。トランスレータは次回用の変数を生成し、今回使用するインスタンス変数を次回用の変数に更新する。更新した値の参照については、更新を行うマクロと同じ制限がある。

例えば、次のようなクローズは、

```
foo([run|Next],Energy,Horse):-
    New_energy:=Energy-100,
    foo(Next, New_energy, Horse).
foo([flee|Next], Energy, Horse):-
    Horse=[run|Next_horse],
    foo(Next, Energy, Next_horse).
```

次のように、記述することができる。

```
class foo(In, Energy, Horse);
run@In->
    Energy/(Energy-100);
flee@In->
    run@Horse;
endclass.
```

この例では、同じ @ による通信の記述を、ガード部（頭部）では受信に、ボディ部では送信に使用している。

通信に使う変数はインスタンス変数であるとは限らない。インスタンス変数の値の中に埋め込まれているストリームを通じて通信する場合には、次回用の変数を指定して、明示的にインスタンス変数の更新を行わなければならない。しかし、FGHC による通信の表記とマクロが混在すると見苦しいので、上のマクロに表記を合わせた図 4 のような疑似ゴールも用意している。また、

(<メッセージ 1>#...#<メッセージ n>)@<今回の通信に使用する変数>

図 3: 通信用マクロ

(<メッセージ 1>#...#<メッセージ n>)
@(<今回の通信に使用する変数>, <次の通信に使用する変数>)

図 4: 次回用の変数まで指定した通信用マクロ

このマクロを使えば、同じクローズの中で、更新されたストリーム変数の値を参照することができる。

ストリームを閉じる場合、あるいは閉じているかどうかを調べる場合は、

```
close(<ストリームを表す変数>)
```

と記述する。

2.5 メソッドセクタ受信用ストリーム

通常、各メソッドがセクタを受信するストリームは決まっている。そこで、セクタ受信用ストリームを、クラスごとに指定できるようにした。これにより、各クローズでのセクタの受信の記述では、そのセクタを記述するだけで済むようになった。

メソッドセクタ受信用ストリームは、class 後のインスタンス変数名リストの中で、from を受信用ストリームとして使用する変数の前に置いて示す。たとえば、クラス foo がインスタンス変数 In をセクタ受信用ストリームとして使用するならば、次のように記述する。

```
class foo(..., from In, ...);
```

このとき、クローズのガード部におけるこのストリームを通じてのメッセージの受信を記述するには、次のようなマクロをクローズの頭部に置く。

```
#<受信しようとするメッセージ>
```

これを使用すると、先に示した例は、次のように記述することができる。

```
class foo(from In, Energy, Horse);  
#run->
```

```
Energy/(Energy-100);  
#flee->  
run@Horse;  
endclass.
```

2.6 メッセージの付加の記述

自分自身にメッセージを送るには、セクタ受信用ストリームの先頭に、送ろうとするセクタのリストを付加する。この操作を記述するために、図 5 のようなマクロを用意した。このマクロは、指定された変数がインスタンス変数であれば、その値を、メッセージ 1 からメッセージ n までを先頭に付加したストリームに更新する。ただし、ストリームがセクタ受信用ストリームで、ソースコード上、このマクロより前で更新が行われていれば、更新された値に、付加を行う。たとえば、

```
class foo(from In, S);  
#one->  
two+In;  
ein@S->  
zwei+S;  
endclass.
```

という記述は、次のような FGHC プログラムに相当する。

```
foo([one|Next_in], S) :-  
    true |  
    foo([two|Next_in], S).  
foo(In, S) :-  
    S = [ein|Next_s] |  
    foo(In, [zwei|S]).
```

最初のクローズでは、セクタ受信ストリームに付加を行っているので、更新された値 Next_in

(<メッセージ 1>#...#<メッセージ n>)+<ストリームとして使用する変数>

図 5: メッセージの付加を行うマクロ

に対して付加を行っている。このマクロが更新した値の参照に関する制限は、他と同様である。

2.7 差分プログラミングのための機能

オブジェクト指向において継承ないし委任による差分プログラミングができることは重要である。しかし、Maglo の機能としては、単一のオブジェクトに対してセレクトの処理を委任する機構しか実現していない。インスタンス変数の継承、多重継承などの、本格的な継承機能は当面の課題である。

委任用ストリーム (委任先オブジェクトのメッセージ受信用ストリーム) を保持するインスタンス変数は、インスタンス変数名リストの中でその変数の前に `to` を置いて示す。例えば、クラス `foo` がインスタンス変数 `Out` を委任先ストリームとして使用するならば、次のように記述する。

```
class foo(...,to Out,...);
```

この指定を行った上で、未定義のセレクトを受け取ったとき、これを委任用ストリームに送信することを指定するためには、図 6 のようなクローズを記述する。

3 記述例

簡単な銀行口座を模したオブジェクトを Maglo によって記述したものを図 7 に示す。このオブジェクトは内部状態として、`Amount` と `Rate` を持ち、`deposit`、`withdraw` などのメッセージに応じて、`Amount` の値を更新する。

また、この口座オブジェクトを利用して、口座に対する操作と同時に、その操作の記録を残すオブジェクトを図 8 に示す。しかし、現在の Maglo は、記録オブジェクトと口座オブジェクトの関係を扱っていないので、図 9 のように、プログラマが共有変数を用意して二つのゴールを同時に起動してやらなければならない。

次に、自分自身へのセレクトの送付を積極的に使用したプログラムの例を、図 10 に示す。これは、ファイルから読み込んだ項から、これを S

式表現に変換したものをファイルに書き出すための命令列を作成するプログラムである。このオブジェクトは、起動時に `read` というメッセージを一回送るだけで、後は自分自身にセレクトを送り続けることによって、主体的に処理を続ける。

また、セレクト受信用ストリームをアクセスすることによって、図 11 のように、簡単に大域脱出などを記述することができる。 `toplevel` では、インスタンス変数 `Exit` に、その時点でのセレクト受信用ストリームの値を退避し、自分自身にセレクト `main` を送っている。 `main` の処理の内部で、自分自身にセレクト `exit` を送れば、`exit` は退避してあった `Exit` の値を新たなセレクト受信用ストリームとして設定し、これにより、制御を `toplevel` に戻すことができる。ある時点で退避されたセレクト受信用ストリームは、その時点に付けられたラベルと見ることができ、退避したストリームを戻す操作はそのラベルへのジャンプと見ることができる。これは、C のライブラリ関数である、`setjmp()` と `longjmp()` の機能と同じものである。

4 効果

Maglo が実現した言語は、およそ、FGHC にオブジェクトを通用範囲とするローカルな述語定義と、このローカルな述語から共通にアクセスできるグローバル変数を加えたものである。また、ストリームの操作の表記を独立させたことによって、通信ストリームを抽象化している。

Maglo によって、オブジェクト指向スタイルのプログラムの記述の手間は、ソースコードの文字数にして約半分になる。また、一つのオブジェクトに所属するクローズに共通する記述を省略し、各クローズに固有の事項のみを記述するようにしたことによって、プログラマの意図が明瞭になり、ソースコードの可読性を大幅に改善している。また、各クローズが受信するセレクトおよび自分自身に送信するセレクトを目立たせることによって、セレクトのローカルな述語としての性格を強調している。

default-><委任と同時に進行すべき処理を記述した本体>;

図 6: 委任を指定するクローズ

```
..., transaction_record(In, Account, []),  
      bank_account(Account, 0, 3/100),...
```

図 9: 記録付き口座オブジェクトの起動

5 関連する言語との比較

これまでもトランスレータによって並列論理型言語に変換することを前提としたユーザー向け言語がいくつか発表されている。以下に挙げた言語は、いずれも、Maglo と同様に、再帰的に呼び出すゴールとその引数の記述を不要としている点で、共通している。また、これらの言語ではオブジェクトの継承など、オブジェクト指向言語に特徴的な様々な機能を備えている。

- Vulcan [Kahn86]
Concurrent Prolog(CP) を基にしており、CP に近い構文を持っている。継承方式としてメソッドコピーと委任の二通りを用意しており、単一継承のみが可能である。メソッドコピーでは、下位クラスのメソッドに、トランスレータが上位クラスのメソッドのゴールを付加する。委任の際は、実行時に、オブジェクトが未定義のセレクトを上位クラスに送付する。
- Mandala [大木 87]
KL1 を基にしており、KL1 に近い構文を持っている。各メソッド呼び出しごとに、メソッドの選択を行う述語を並列に呼び出すことによって、オブジェクト内で並列処理を行っているのが特徴である。インスタンス変数のアクセスの相互排除を行うために、専用のオブジェクトにインスタンス変数の値を持たせ、このオブジェクトとの通信によってアクセスを行うようになっている。継承は、上位クラスのメソッド選択述語を直接呼び出すことによって実現している。多重継承が可能である。
- A'UM [Yoshida87]
KL1 を基にしており、Smalltalk80 風の構

文を持つ。整数を含めてあらゆるデータを一律にオブジェクトとして扱っており、したがって、インスタンス変数のアクセスはすべてストリームを介して行う。ストリームで表されたオブジェクト同志の同一化もサポートしている。オブジェクトの内部での主な処理は自分自身へのメッセージの送信に展開され、したがって具体的な処理はメッセージが受信された順に逐次行われる。継承は、一時的に所属クラスを上位クラスに変更することによって実現しており、多重継承が可能である。A'UM のプログラムは、少数の決まった述語だけからなる KL1 プログラムに変換される。これらの述語を仮想命令として持つ仮想機械は、KL1 よりも効率よく実装できる可能性があり、これによって A'UM の効率のよい実行が期待できる。

他のトランスレータと Maglo が最も異なっているのは、次の二点である。

- オブジェクトでないデータを直接扱っている。
A'UM では、すべてのデータはオブジェクトであり、ストリームを介してのみアクセスできる。また、Mandala では、オブジェクトでないデータをインスタンス変数の値として扱うことができるが、インスタンス変数は、専用のオブジェクトの中に保持されており、これとの通信によってアクセスを行う。Maglo では、オブジェクトでないデータは直接 FGHC の変数の値としてアクセスできるので、通信によるオーバーヘッドを回避することができる。
- セレクト受信用ストリームを陽にアクセスできる。

```

class account(from In, Amount, Rate);
close(In)-:
    true;
#amount(X)->
    X=Amount;
#rate(R)->
    R=Rate;
#add_interest->
    Rate=N/D,
    Amount/(Amount+(Amount*N/D));
#deposit(X)->
    Amount/(Amount+X);
#withdraw(X)->
    Amount/(Amount-X);
default->
    true;
endclass.

```

図 7: 口座オブジェクト

```

class transaction_record
    (from In, to Account, Record);
close(In)-:
    close(Account);
#deposit(X)->
    deposited(X)+Record,
    deposit(X)@Account;
#withdraw(X)->
    withdrawn(X)+Record,
    withdraw(X)@Account;
#add_interest->
    (amount(Old)#add_interest
    #amount(New))@Account,
    I:=New-Old,
    interest_added(I)+Record;
#record(X)->
    X=Record;
default->
    true;
endclass.

```

図 8: 記録付き口座オブジェクト

```

class sexp(from In, Console, to Out);
#read->
    read(X)@Console,
    (eof_check(X)#read)+In;
#eof_check(end_of_file)-:
    close(Console),
    close(Out);
#eof_check(X)->
    otherwise|
    (term(X)#nl)+In;
#term(X)->
    X\[ _|_ ]|
    functor(X,F,N),
    atom_or_str(X,F,N)+In;
#term([CAR|CDR])->
    % (. <CAR> <CDR>)
    (put(40)#put(46)#put(32)
    #term(CAR)#put(32)
    #term(CDR)#put(41))+In;
#atom_or_str(S,F,N)->
    N=0|
    write(S)@Out;
#atom_or_str(S,F,N)->
    N\[ 0|
    % 40 = "("
    (put(40)#write(F)
    #args(S,1,N))+In;
#args(S,I,N)->
    I<N|
    arg(I,S,A),
    % 32 = " "
    (put(32)
    #term(A)#args(S,K,N))+In,
    K:=(I+1);
#args(S,N,N)->
    arg(N,S,A),
    % 41 = ")"
    (put(32)#term(A)#put(41))+In;
default->
    true;
endclass.

```

図 10: S 式への変換を行うオブジェクト


```

class foo(from In, ..., Exit);
#toplevel->
    Exit/In,
    main+In;
...
#exit->
    In/Exit;
endclass.

```

図 11: 大域脱出の記述

Vulcan と Mandala では、セレクト受信用ストリームは処理系の中に埋め込まれていて、このストリームからセレクトを受信することだけが可能である。A'UM では、`$self` という名前でも参照できるが、このストリームの更新は、自分自身へのメッセージの送付という形でのみ可能である。Mango ではセレクト受信用ストリームの扱いは、原則としてインスタンス変数と同じであり、自由に参照と更新を行うことができる。セレクト受信用ストリームのアクセスによって、プログラム例で示した大域脱出のような、制御の流れの変更を記述することができる。

6 問題点

FGHC で直接オブジェクト指向スタイルの記述を行うのに比べれば現状でも記述効率は大きく改善されており、当初の目的は達しているが、それでも実際にプログラムを記述していると、不便を感じるものがいくつかある。

小さなプログラムを書いているだけでも気がつくのは、手軽な制御構造がないことである。条件分岐は FGHC のガード部として、ループはローカル述語の再帰呼び出しとして記述しなければならない。これらの構造を一つ作るたびに、そのためのセレクトを一つ定義しなければならない。たとえば、図 10 の `eof_check` というセレクトは、引数が `end_of_file` であるかどうかを調べるためだけに定義したものである。新たなマクロを設定すればよいのだが、プリミティブなマクロを増やすよりは、ユーザーがマクロを定義できるようにすることで対処したいと考えている。

また、継承機能の不備も挙げられるが、これ

は、他のトランスレータを参考にすれば、取るべき手段は比較的明らかであり、できるだけ早く装備したいと考えている。

7 謝辞

様々な援助をくださった和田英一教授および和田研究室の皆様にお礼を申し上げます。

参考文献

- [Yoshida87] K. Yoshida, T. Chikayama, *A'UM-Parallel Object-Oriented Language upon KL1*, ICOT Technical Report: TR-308, 1987.
- [大木 87] 大木 優, 竹内 彰一, 古川 康一, 並列論理型プログラミング言語 *KL1* を基にしたオブジェクト指向プログラミング言語, Proc. of Logic Programming Conference '87.
- [Kahn86] K. Kahn, E. D. Tribble, M. S. Miller, D. G. Bobrow, *Objects in Concurrent Logic Programming Languages*, OOPSLA '86 Proc.