バインダによるモジュール間アクセスの制御

3N－4

杉山 潤，越田 一郎
東京工科大学†

1　はじめに

筆者らは，拡張性および柔軟性の高いプログラミング環境を構築するための一手法として，「プログラム情報指示子」という機能を持つ言語を提唱した[1]．本稿では，これを拡張した「バインダ」によって，モジュールのグループ単位で，モジュール間の関数呼出しを制御する方法と，バインダによるデバッギング環境などへの応用について述べる．

2　バインダとグループ

2.1　プログラム情報指示子の改良

プログラム情報指示子では，関数呼出しを含む任意の式を，ステートメントなどに結び付けることができる．しかし，結び付けられた各関数は，それぞれの関数内で実行すべきコードを切り換えたり，無効にすることはできたが，新たに処理を追加・変更するためには，その部分を書き換える必要があった．

関数本体のコードを書き換えることなく，その処理内容を切り換えるには，関数の外部から切り換えの処理を行うためのメカニズムが必要である．それを具体化したものとして，切り換えの最小単位であるモジュールと，制御の煩雑さを減少するためのグループ，及びグループ間の呼出しを記述するためのバインダによる方法を提案する．

2.2　グループとは

プログラムを構成するモジュールはその機能別に分類して，グループ名を付けておき，そのグループを単位として，モジュール間アクセスの制御を行う．アクセスの制御は，呼出し側のモジュール内に，プリミティブ関数を記述することによって行う．グループは，以下に示す特徴を持っている．

- グループの生成は，プリミティブ関数によって行う．
- グループの削除は，生成を行ったモジュールの関数内で行う．
- グループは，それに含まれるモジュールとサブグループの名前を管理する．
- 親グループのモジュールから，サブグループを一時的に無効にできる (ON/OFF 制御)．

†Inter-module access method with a binder
Jun Sugiyama, Ichiro Koshida
Tokyo Engineering University,
1404-1, Katakura, Hachioji, Tokyo, Japan

2.3　バインダとは

同じグループ内では，従来と同様の静的な関数コールが利用できるが，異なるグループ間では，バインダで関数名を指定することで行う．バインド機能は，以下のように動作する．

- バインド元のグループのサブグループに含まれるモジュールに，バインドされた関数が存在するとき，その関数は呼び出される．
- バインド先の関数から，バインド元のモジュール内の変数などを読み込むことができる．書き込む場合には，その変数を関数の引数で渡す．
- バインド元の処理とバインド先の関数は，並行に動作する．そして，両方の処理が終了したら，次のステートメントの処理に移る (図 1)．

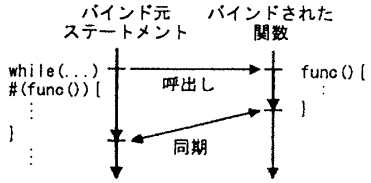

図 1. バインドされた関数の実行

2.4　グループとモジュール

モジュール内のコードは，プログラマが書き換えない限り変化することはない．それに対して，グループとそれに含まれるモジュール名は，プログラムの実行に応じて動的に変化し，実行されるコードがモジュール単位で切り換えられる (図 2)．

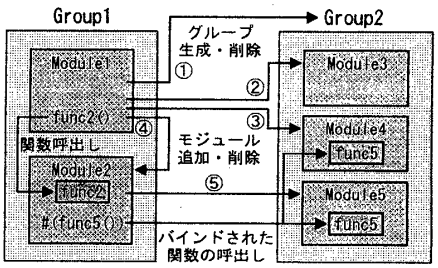

図 2. グループとモジュールの動作

2.5　バインダを使う利点

バインダによって，呼び出すモジュールの切り換えが動的に行えるだけでなく，同じような内容の処理を行うモジュールをグループ化することで，制御の煩雑さが軽減される．

3 バインダをサポートする言語

現在，バインド機能を持つ言語のインタプリタを作成している．その構文は，C言語やPascalなどに近いものである．ここでは，バインド機能に関係のある部分について説明する．

3.1 バインダの書式

異なるグループに属するモジュール内の関数を呼び出す際には，バインダによって関数やステートメントに結び付けを行う．

<ステートメント等># (*expression*)

ここで *expression* には，バインド先のモジュール内の関数を呼び出すための式を記述する．

3.2 バインド機能の実装

グループ内の関数は，ハッシュテーブルにアドレスを登録しておき，呼出し時にはそれを検索すればよい．しかし，グループ外の関数は，その時点でのグループの状態によって，呼び出すべき関数が変わってくる．そこで，グループごとに，モジュールの名前を登録された順序に従ったリストに記録しておき，呼出しの際には，最後に登録されたモジュールから順に関数を検索し，見つかり次第それを呼び出す．

4 バインダを用いたデバッギング環境

バインド機能を利用すると，バインド元のスコープでアクセスできる変数がバインド先でも読めるため，デバッグなどに便利である．

4.1 プログラムの表示形式の変更

プログラムのコードやデータは，それぞれの使われ方や意味だけでなく，プログラマの好みによっても，最適な表示形式が異なってくる．それらすべての表示形式をあらかじめ用意することは，事実上不可能であるので，追加モジュールの形で開発環境の拡張を行う．

4.2 デバッグコードの制御

あるモジュールで，デバッグ済みのモジュールを使用する場合，デバッグ済みの部分に関する情報は，そのほとんどが必要なくなる．ただし，実行中にエラーなどによって特殊な状態に陥ったときに，それらのデバッグコードが役立つこともある．この場合，エラーを検出した時点で，デバッグコードの一部を有効にすることで実現できる．

4.3 プログラム例

実行環境によってエラー出力の方法を切り換える，簡単なプログラム例を以下に示す．

```
 1 module Main;
 2 main(){
 3   MakeGroup( "ErrOut" );
 4   AddModule( "ErrOut", "ErrOutStd" );
 5   AddModule( "ErrOut", "ErrOutLog" );
 6   AddModule( "ErrOut", "ErrOutGui" );
 7   while(){ // main loop
 8     ...
 9     if( Error() ){
10       if( FileError() )
11         DeleteModule( "ErrOut", "ErrOutLog" );
12       if( GuiError() )
13         DeleteModule( "ErrOut", "ErrOutGui" );
14       ErrorRecovery() #( PrintErrMsg( Msg ) )
15 } } }
16 endmodule;
17
18 module ErrOutStd; // output message to stderr
19 PrintErrMsg( s ) { ... }
20 endmodule;
21
22 module ErrOutLog; // output message to log file
23 PrintErrMsg( s ) { ... }
24 endmodule;
25
26 module ErrOutGui; // output message to window
27 PrintErrMsg( s ) { ... }
28 endmodule;
```

リスト1. エラー出力コードの切り換えプログラム

5 まとめ

グループとバインダによって，新たな構造化レベルと，それに対するアクセスの手段を提供することができた．さらに，コードの動的な切り換えとバインダによる並行性の記述を利用することで，分散環境への応用も考えられる．

今後は，細部の仕様を煮詰めていくと同時に，この言語のための開発環境を構築し，その際に必要と思われる言語機能やプリミティブ関数を追加していく予定である．

参考文献

[1] 杉山 潤, 越田 一郎: 制御情報の埋め込みが可能な言語の作成とビジュアルプログラミングへの応用, 情報処理学会 第51回全国大会 講演論文集, 3L-7, 1995

[2] 所 真理雄, 松岡 聡, 垂水 浩幸: オブジェクト指向コンピューティング, 岩波書店, 1993

[3] Dylan Interrim Reference Manual, 1994