Continuation based C を使ったソースコードのリファクタリング手法

Refactoring method using Continuation based C

河野 真治

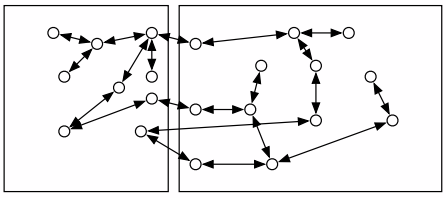
Shinji Kono E-Mail: kono@ie.u-ryukyu.ac.jp *

Real-time や Interactive なプログラムでは、ソフトウェアに必須な再構成は、分割された階層にそって生じるとは限らない。ここでは、Real-time や Interactive とオブジェクト指向プログラムについて考察し、継続を用いたプログラム言語 CbC によって、それらの再構成がどのように可能になるかについて考察する。

1 なんのために refactoring するのか?

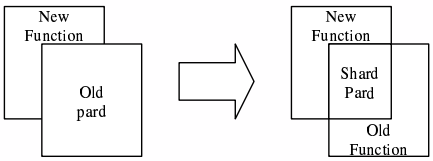
ソフトウェアは一気に構築されるわけではなく、幾つかの変更を得て作られていく。新しい環境で使われるように変更することもあれば、再利用しやすくするために同じ動作をするプログラムを異なる構成に変更することもある。これらの再構成がリファクタリングと呼ばれる手法である。

人間が作るソフトウェアは、いかに巨大なものであっても人の手で修正できる範囲の単位に分割されて作成される。しかし、プログラムのステートメントは、変数や I/O にアクセスするだけであり、それらの動作と分割には直接の関係はない。従って、あるソフトウェアに対して自明な最適な分割というものは存在せず、ソフトウェアの構築の履歴にそってアドホックな分割が行われることになる。



ソフトウェアの初期のプロトタイプに分割と、中期後期の改良や機能の追加、メンテナンスに向いた分割はおのずと異なり、ソフトウェアの再構成が必要なることが多い。

特に機能の追加を行う場合は、新しい機能は古い部分の一部の機能を利用することが普通である。その古い部分が再利用可能な形に前もって設計されている場合ばかりではないので、その部分を、再利用しやすい分割への変更する必要がある。



機能の追加は、単純なコードの追加ばかりでなく、速度、スペース、省電力などに対する最適化が要求されることもある。この場合は、局所的なコードの変更で対処できずに、全体的な変更が必要となることがある。

オブジェクト指向やモジュール化は、通常は、間接呼び出しを用いた動的な呼び出し機構を要求する。これらは、PC 上のアプリケーションのような機能に比べて CPU の能力が高い場合には問題ないが、組込みシステムや携帯電話のようなシステムでは、そのオーバヘッドが問題になることがある。動的な呼び出しと静的な呼び出しのバランスを見直すこともリファクタリングの一つである。

2 継続

継続とは、計算の続きを示すプログラムの機能のことである。継続は、プログラミング研究の比較的初期から導入されていたが、プログラミング理論やコンパイラ理論などで、その有効性 [1] が再確認されている。現在のプログラミング言語では、例外処理や大域脱出の機能として実装されていることが多い。

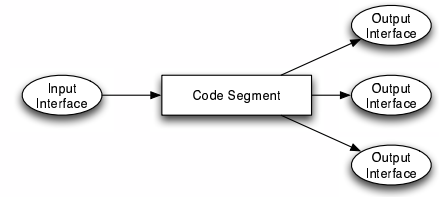
現在広く使われているプログラミング言語では、関

*Information Engineering, University of the Ryukyus,

数呼び出しをスタック上に実現した環境の入れ子で表すのが普通であり、継続は、そのスタックへの隠れたポインタを持つのが通常である。このスタックは継続の呼び出しや保存に従って適切に処理する必要があり、そのせいで Scheme や C++ あるいは Java の継続は極めて重い機能、あるいは、制限された機能となっている。

3 Continuation based C

筆者らが提案している言語 Continuation based C [3, 2] は、環境を持ち歩かない継続を持つ言語であり、C からサブルーチン呼び出しを取り除いたコード・セグメントという単位を持つ。コードセグメントには、入口に相当する Input interface と、出口に相当する Parameterized goto 文が存在する。



以下は簡単な CbC のプログラムである。

```
code fact(int n,int result,
       code print()){
    if(n>0){
        result *= n;
        n--;
        goto fact(n,result,print);
    } else
        goto print(result);
}
```

間接 goto と、直接 goto が、プログラムの単位の結び付きをボトムアップに規定して、自然なグループを構成する。

4 C から CbC への変換

CbC には、サブルーチン・コールを含む上位言語 CwC(C with Continuation) があるが、C の関数呼び出しのスタックベースの環境を明示的にすることにより、C から CbC への変換が可能である。

```
j = g(i+3);
```

のような C の関数呼び出しは、`struct f_g0_save` などの明示的なスタックの中身を表す構造体を用いて、

```
struct f_g0_interface *c =
  (struct f_g0_save *)(sp -=
  sizeof(struct f_g0_save));
c = sp;
c->ret = f_g1;
goto g(i+3,sp);
```

のような形で、明示的なスタック操作に変換される。これは変換の一例であり、他の方法、例えばリンクトリストなどを用いても良い。`f_g1` は、関数呼び出しの後の継続であり、`g` では、

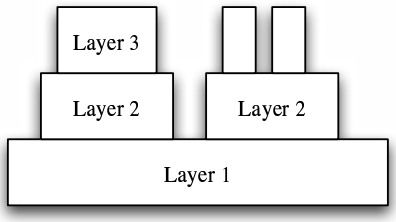
```
code g(int i,stack sp) {
    goto (* ((struct
     f_g0_save *)sp)->ret)
       (i+4,sp);
}
```

のように間接的に呼び出される。スタックの中は、継続と中間変数などを格納する構造体である。スタックそのものは、これらの構造体を格納するメモリである。

しかし、例えば、`sin(x)` のようなサブルーチンを無理に goto 文に変更する必然性がなければ、その部分を変換しなくて良い。

5 階層的プログラム分割の限界

サブルーチン・コールは、呼び出しのところに戻るための実装を持ち、それを中心したモジュール構成は、必然的に階層的なプログラム分割を持つ。



ライブラリやツールキットは、必ず階層的な構成を持つ。ネットワーク階層などや複数のルックアンドフィールをサポートする GUI などの場合は、下位階層が複数から選択される場合があり、そのような場合は、間接呼び出しを経由することになる。

分割された部分は完全に独立に動作するわけではなく、要求された機能によって、上位から下位へのアクセスがあり、外部インベトがあれば、下位から上位へのアクセスがある。これらは、リフレクション的な手法で実現される。プログラム言語的には、言

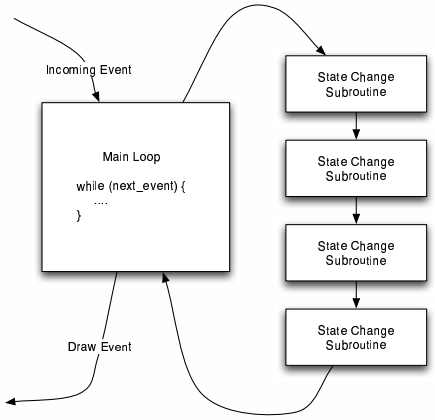
語機能外の API となっていることが多い。例えば、Unix ならば、ioctl などがそれに相当する。

しかし、Real-time プログラミングや、Interactive プログラミングでの要求される仕様は、必ずしも階層的な分割にしたがったものとは限らない。例えば、レスポンス時間に対する制約は、ネットワーク階層全体に依存する。階層的な分割は、より多くの要求に対して適切に対応できるように設計されるが、万能な階層分割は存在しない。

これらのプログラムは、単純な入出力ではないので、それに対応した構造を持つ。良く使われる構造について次の節で考察する。

6 Real-time/Interactive Programming

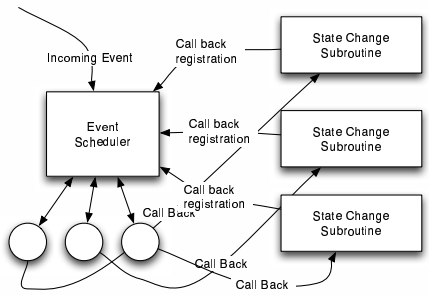
Real-time/Interactive プログラミングのもっとも基本的な構造は、入力イベントを待ち合わせるメインループを持つものである。



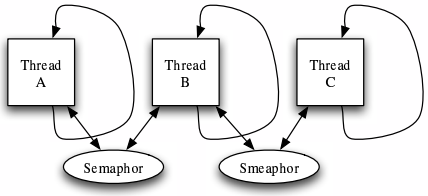
テレビゲームあるいは、下位ライブラリ、または、OS の中核でも、このようなループが現れる。原始的な手法であるが、これが基本であり、すべての制約をループで処理することができる。

イベントによる状態遷移はソフトウェアの様々な部分で起きる。したがって、その変化を部分品に配布する手法が必要となる。静的に固定されたソフトウェアでは、順に呼び出せば良いが、動的に変換する部品の場合は、変更を担当するルーチンを登録する必要がある。これらのルーチンはコールバックと呼ばれる。

コールバックは部品の独立性を高めることが出来るが、コールバックのつながりは低い。また、部品が自律して動作するような記述には向かない。



そのような場合には、スレッドを用いた記述を使うことができる。スレッドは、独自なメインループを持つのが普通であり、外部イベントまたは他のスレッドとの同期を待って処理を行う。前の二つよりも部品独自の状態遷移記述を独立的に記述できるが、スタックやスケジューラ等の資源を要求する記述でもある。



7 同期を経由したアクティビティの移動と通信

これらの三つの構造 (ループ、コールバック、スレッド) は、相互に補完するものであり、動的に拡張される部分や要求される応答速度、資源の制約などによって選択される。しかし、この構造はモジュールやオブジェクトの持つ階層的な構造とは直交しているので、この三つの関係を相互に変換するのは一般的には難しい。

例えば、セマフォを通して同期しているスレッドは、他方がセマフォを解放すると、他方が動き出すと言う逐次実行の関係にある場合がある。しかし、これをサブルーチン呼び出し、あるいは、コールバックで実現することは難しい。

一方で、メインループを使ってスレッドやコールバックを実現することも可能だが、独自のライブラリを自分で構築するような手間をかけることになる。

ソフトウェアは常に動的に変化しているわけではなく、ある時間を見ると定常的な動きをしているこ

とが普通である。例えば、ブラウザのダウンロード中は、ダウンロード状況を示すアニメーションを表示する定常的な動作を行えば良い。これらの定常的な動作の負荷を下げるためには、動的なコールバックやスレッドを避ける方が良い。しかし、動的な変化に対応するためには、複雑なデータ構造と間接呼び出しを経由しなければならない。

8 オブジェクト指向との関係

Real-time/Interactive プログラミングでも、ソフトウェアはなんらかのオブジェクト指向的な構造を持つが通常である。オブジェクトは、メッセージ・パッシングで相互作用するが、現在のメッセージ・パッシングは間接コールによって実現されているのが通常である。

本来は、オブジェクトは自律的に動いているもので、その間のメッセージ・パッシングは、メッセージボックスなどを経由したプロセス間通信である。オブジェクト指向言語の元になった Simula では実際にそのように実装されている。しかし、プログラム言語としての Smalltalk や、その後継言語である C++ などでは実装の軽さからメッセージ・パッシングが採用されている。

メッセージは、オブジェクトに対して送られるので、本来は、そのオブジェクトの持つ情報のみによって計算と状態変化が進む。しかし、サブルーチン・コールは、環境を持ち歩くので、その環境から来る状態がオブジェクトに追加されてしまっている。

9 継続を用いたオブジェクト指向

メインループのような手法の場合は、メッセージパッシングは、その呼び出し元に戻る必然性はなく、メインループまたは、後続する状態遷移のためのオブジェクトへのメッセージパッシングをおこなえば良い。このような場合は、CbC の goto 文でメッセージパッシングを十分に実現することができる。

```
    code a(object *self,event *e){
        if(my_even(e)){
            state_change(self);
        }
goto self->continue(self->next,e);
    }
```

この場合は、必要な呼び出しをスケジューラから間接的に行うのではなく、コード・セグメントの間接呼び出しの接続で行う。

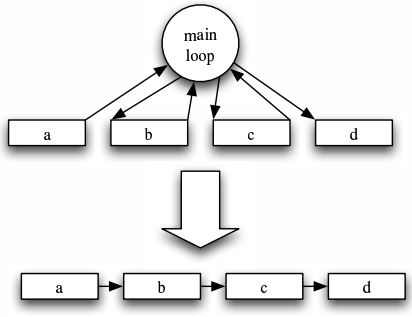
10 継続を用いたオブジェクトの分割と再構成

リファクタリングの一部として、オブジェクトの分割がある。機能追加などによる共有部分を独立させる作業がオブジェクトの分割に対応する。

機能はメインループやコールバック、スレッドの動作の順序にしたがって呼び出され、その共有部分を別なオブジェクトとして抜き出すことになる。メッセージパッシングとして、CbC を用いると、関数呼び出し順は重要ではないので、その順序にしたがって切り出せば良い。

ただし、CbC では、コードセグメントで必要なすべてのデータは、Input Interface にそろっている必要がある。Input Interface に、オブジェクトへのポインタがあれば、通常は、そのポインタ経由でオブジェクトの状態変更に必要な情報はすべて手に入る。

関数呼び出しとして実現されているメッセージ・パッシングの場合は、メソッドを部品として分割する場合には、そのメソッドを呼び出す主体を考慮した分割が必要となる。コールバックやスレッドは、その呼び出し関係を単純にする役割があるが、コードセグメントの呼び出し関係を変更する手法では、これらの三つを対等に扱うことができる。例えば、メインループ方式から、直接に固定された状態遷移を呼び出す逐次的な呼び出しに変換することが容易となる。



ただし、これらのコードセグメントの接続は、大域変数または、スレッドを用いる場合は、スレッド・ローカルな領域に記述する必要があり、その手法に関しては、現在、さまざまなアプリケーションに適した手法を開発しているところである。

11 まとめ

本論文では、Real-time/Interactive プログラミングとオブジェクト指向言語の関係について考察し、継続を基本とする CbC 言語により、これらを記述することにより、オブジェクトの分割をより自由に行う方法を提案した。

参考文献

[1] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[2] 河野 真治. 継続を基本とした言語 CbC の gcc 上の実装. 日本ソフトウェア科学会第 19 回大会論文集, Sep 2002.

[3] 河野 真治. 継続を基本とするプログラム単位を用いた組込みシステム開発. 組み込みソフトウェア工学シンポジウム 2003, Oct 2003.