21. 多重環境の制御実現法†

佐治　信之††

1. はじめに

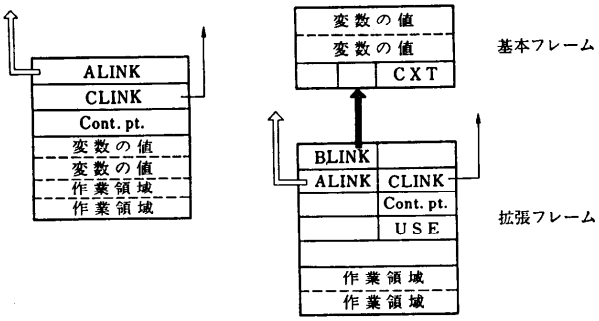
「環境」という言葉から連想されるものは，実に多いわけであるが，ここでは「プログラムの実行のための環境」を対象に話を進めることにする．

プログラムをいくつかのモジュールに分割した際，各モジュール間に，一意の従属関係が見い出せない場合がある．たとえば，探索一選択という問題を考えるとき，探索と選択の2つのプロセスは，どちらかが他方に属している．つまり一方をサブルーチンと考えるよりは，各々独立なルーチン，即ち，**コルーチン**と考える方が自然であろうし，また問題の本質を，的確に表現していると言える．しかし，探索と選択の各プロセスがコルーチンを構成するためには，探索に必要な環境（走査がどこまで進んだか，など）も，選択に必要な環境（過去の選択状況など）も，同時に保持されていなければならず，元の問題が自然に表現された代りに，今度はこれらの環境を，どう効率よく保持するかという，新しい問題が生まれてくるわけである．

本解説では，コルーチン実現などで必要とされる複数の環境を**多重環境**と呼び，この多重環境の効率的な管理法について述べる．

2. スパゲティ・スタック

1973年に，Bobrow と Wegbreit は，多重環境の管理法として，**スタック**を一般化したモデル[3]を発表している．これは**スパゲティ・スタック**と呼ばれている．Wegbreit 自身は，これ以前に，2-スタックによる，バックトラッキングの実現法[11]を発表していたが，スパゲティ・スタックのモデルは，バックトラッキングばかりでなく，コルーチンや，関数引数の処理なども実現可能な，拡張されたものになっている．スパゲティという名は，実行中にもしポインタが線として見えるなら，スパゲティのように複雑に絡み合っているだろう，ということで付けられたようである．



(a) 再帰呼び出し用フレーム　(b) 一般化されたフレーム

図-1

2.1 スタックとフレーム

Pascal などでは，プログラムの実行の際に，**図-1**(a)のような「**環境フレーム**」を関数（手続き）単位で割り付けて，再帰呼び出しを実現している．alink は，変数の参照の際にたどるべき親の環境フレームを指すリンクである．clink は関数の評価が終ったときに制御を戻すべき親の関数の環境フレームへのリンクであり，このとき，親の関数の評価がどこから（何番地から）再開されるかが，cont. pt. に入れられている．

スパゲティ・スタックモデルでは，この環境フレームが，**図-1**(b)のように，より一般化されている．関数（手続き）単位に，基本フレームと拡張フレームという2つのフレームが割り付けられる．これらは，(a)の環境フレームを2つに分けたと思えばよい．そして，2つのフレームは，blink と呼ばれるリンクで結ばれている．alink，clink は（a）のものと同一の目的で使用されるが，cont. pt. の使い方は若干違いがある．このフレーム構造で特徴的なのは，基本フレーム中の CXT と拡張フレーム中の USE である．

† Control Implementations of Multiple Environments by Nobuyuki SAJI (Soffware Product Engineering Laboratory NEC Corporation).

†† 日本電気(株)ソフトウェア生産技術研究所

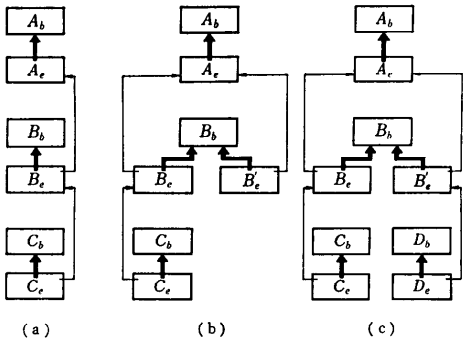


図-2

これらは，フレームの**参照カウンタ**であり，ゼロになったときに，そのフレームが削除される仕組みになっている．

さて，このフレーム構造を用いて，コルーチンは，どのように実行されるのだろうか．(**図-2**) まず，A_b，A_e は，それぞれ関数 A の基本フレーム，拡張フレームを表わす．関数 A が関数 B を呼び，関数 B が関数 C を呼んだあと (a)，関数 C の実行を中断して，関数 B に戻り (b)，関数 D を呼び出した時のスタックの状態が，図-2(c) である．以降，関数 C と関数 D はコルーチンの振舞いをするわけである．

ここで注意せねばならないのは，次の2点である．

(i) 実行を中断した関数 C が，実行を再開して関数 B に戻るときの B 中の番地と，関数 D が関数 B に戻るときの B 中の番地は異なる．

(ii) 関数 B 中のある式の項として関数 C が呼ばれていたとすると，B_e 中には，式の他の項の評価結果が置かれている可能性がある．

すなわち，関数 C と D が必要とする関数 B の環境は，別々に管理されなければならない．それで 図-2 (c) では，関数 C と D の各々の環境フレームに対して，B_e, B_e' という2つの拡張フレームが割り当てられている．一方基本フレームは，関数 C と D で共有しており，基本フレーム中の変数を利用することで，関数 C と D の間のデータの受け渡しや，その他の制御を実現している．これで，フレームが2つに分かれている理由が明らかになったであろう．

スパゲティ・スタックモデルでは，図-1(b) のフレームを，スタック上に割り付ける．完全な LIFO (last-in first-out) の規則に基づくスタックでは，通常の再帰呼び出しは処理できても，コルーチンなどは管理ができない．たとえば，2つのコルーチンは，どちらかのフレームがスタック上に先に割り付けられるが，先に割り付けられた方のコルーチンが後のものより早く終了してしまうと，スタック中に使用していない部分，つまり穴があいてしまうからである．使用後，フレームを開放するには，USE と CXT の2つの参照カウンタを用いるがこのフレーム再生の詳しいアルゴリズムについては，原論文[3]か解説[16]を参考にしていただきたい．

2.2 圧　縮

前項で，コルーチンの場合に，スタックに穴があくことを示したが，これにはもう一つの可能性がある．実は，拡張フレームは，関数が起動されて割り付けられるときに，式の評価の際の中間結果を保持しておくための作業領域を一切持っていないので，実行中にフレームの大きさが変わるのである．ここで 図-2 の例を思い出してほしい．関数 D の実行を中断し，関数 B に戻った時に，B_e' フレームは，D_b にブロックされており，十分な作業領域を確保できない恐れが出てくる(**図-3**)．この場合には，他の空き領域に B_e' 全体をコピーしてから続行することになる．拡張フレームのコピーの割り付けには，次の2つの方法が考えられる．

(i) 常にコピーはスタックの先頭に割り付ける．スタックがあふれたら，圧縮を行い，領域を再生する．

(ii) 空き領域をリストにしておき，リストから取り出して割りつける．(依然として圧縮は必要) ちな

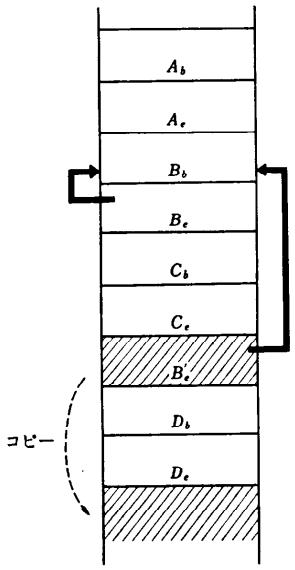


図-3

みに，原論文では（ｉ）が採用されている．

3. その他のモデル

スパゲティ・スタックが実際に使われている例としては，INTERLISP[14] が有名である．コルーチンやバックトラッキングが可能な言語としては，SIMULA 67[4]，GEDANKEN[12]，CONNIVER[9]，SL 5[6]，Lisp Machine Lisp[15] などがあげられる．このモデルをファームウェア化して Lisp に組み込んだ例[18]もあるが，あまり使われていないというのが現実のようである．完成度が高く，ある程度の効率も保障されているとはいえ，やはり問題点もあり，このままでは採用されるに至っていないようである．この節では，スパゲティ・スタックのモデルを基本にした，種々の改良案について触れることにする．

3.1 マカロニとスパゲティ

Steele[13]は，スパゲティ・スタックの問題点を次のように指摘している．

（ｉ） スパゲティ・スタックは，動的結合規則*を基礎にしているが，静的結合規則**を基礎にすべきである．（詳細は原論文[3]を参照）

（ⅱ） **FUNARG***** 生成の際にスパゲティ・スタックでは，拡張フレームまで保持しているが，不要である．

（ⅲ） （ⅱ）と同様に，alink は親の拡張フレームを指す必要はなく，基本フレームを指せばよい．

（ⅳ） スタック１本で，フレームを管理するのは，効率上限界がある．

これらの問題点を改良してつくられたマカロニ・スタックのモデルは次のようになる．

a）環境フレームは，アクセスフレーム（基本フレームに対応）制御フレーム（拡張フレーム）動的結合フレーム（拡張フレーム）の３つで構成される．

b）alink の格納場所は，拡張フレームから基本フレームへ移動し，親の基本フレームを指す．

c）アクセスフレーム用と制御フレーム用の２本のスタックを用意する．

マカロニがスパゲティより優れていることを示す顕著な例に，upward funarg**** がある．図-4 では，

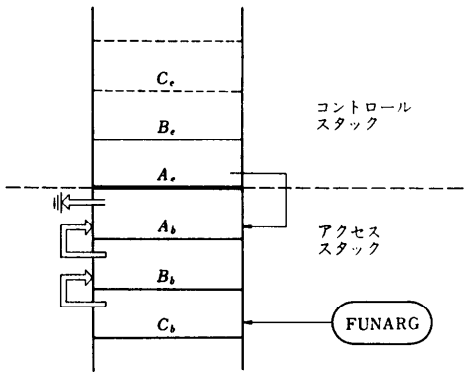


図-4

関数AがBを，BがCを呼び，Cが関数Dを値としてBに返し，Bが同じ値をAに返した時のスタックの状態である．関数Dは，関数 B, C の変数と値の結合状態を保持しておく必要があるが，それ以上の情報は不必要である．つまり，関数 B, C の拡張フレーム B_a, C_a は，保持しておかなくてよい．b），c）の改良が効果的であることがわかる．

3.2 スタック・グループ

Greenblatt[5] は一本のスタックでの管理の限界を指摘して，複数のスタックの集合である**スタックグループ**のモデルを提案し，Lisp 処理系で実現している[15]．

（ｉ） 環状バッファをいくつかに分割して，各々を，スタックとして使う．全体はスタックグループである．コルーチン実行の際には，別々のスタックに割り付ける．

（ⅱ） あるコルーチンの消滅は，使用されていたスタックの上下の境界を消すだけである．

東出らの Lisp[19] の実現には，このスタックグループのモデルが使われており，さらに，基本フレーム中の変数と値の結合表を Lisp のリスト領域に追い出すなどの工夫が見られる．

3.3 スタック＋ヒープ

Berry らはブロック構造を持つプログラム言語を対象とした，環境保持のモデルを２つ発表している．これらはいずれも，言語の提供するデータ型を，環境保持の面から分類して，きめの細かい処理を行うことで効率向上を試みているもので，スタックとヒープを併用している．２つのモデルの相違点は，フレーム管理の方法が，参照カウンタ法[1]か，３パスのゴミ集め[2]かの違いである．以下に，参照カウンタ法のモデルの特徴をあげておく．

---

* 実行中の環境をたどり，変数の値が何であるかを知る．Lisp 1.5 の a-list による評価規則．

** Pascal や Algol で採用されている規則．

*** 関数へのエントリと，環境フレームへのポインタの組で，データオブジェクトの１つである．

**** FUNARG オブジェクトが，関数の結果値として返される場合を言う．

(i) contour セグメント (拡張フレームの制御情報+基本フレーム), 評価スタック (作業領域として共用), スタックセグメント (評価スタックのコピーを保持) の3つで構成される.

(ii) 各変数は型を持ち, 環境保持に関連する型 (ラベル型, ポインタ型など) の変数は contour セグメントへのポインタを持ち, contour の中にある参照カウンタの更新を行う.

ゴミ集め法のモデルは, これより若干単純な構成であるが, いずれにしてもこれらの複雑さは, スパゲティの比ではないことを記しておく.

4. 性能評価

スパゲティ・スタックと, そのさまざまな改良案との性能評価を行っている論文を2つあげておく.

4.1 スパゲティ vs マカロニ

文献10) の評価は, 16のサンプルプログラムの実行において使用した, スタック領域の最大値と, フレームのコピーの回数を, 比較基準にしている.

評価の結論は,

(i) コルーチン同志が直接呼び合う場合

a) 空き領域リストからフレームを割り付ける.
(ただし last freed first filled の規則で行う.)

b) フレーム構造はマカロニ方式

c) スタックはアクセス/制御用の2本を用意

以上の方法の組合せが最良であった.

(ii) セミコルーチン* の場合

a) 常にスタックの先頭にフレームをコピー

b) フレーム構造はマカロニ方式

c) スタックは1本用意

以上の組合せが最良であった.

となっている. マカロニはスパゲティよりも優れている, という原論文のタイトル[13]は真実を語っているようである.

4.2 ヒープ and/or スタック

文献7) では, 素朴にヒープのみを用いる方法, スパゲティ・スタックを用いる方法, そしてスタックとヒープの併用法の3つの方法の比較を行っている. ただし, ここでのスタック・ヒープ併用法は, 前述の Berry 流ではなく, この論文の著者達の提案しているモデルである.

この性能評価では, 延べメモリ使用量を総合的な評価規準として比較が行われており, また前述の拡張フレームの再割り付けに関しても, さまざまな場合が比較に組み込まれていて, 興味深い.

結果をあげると次のようになる.

(i) 拡張フレームの作業領域をあらかじめ確保しておく方法の方が, そうでないものより 30%～300% 効率が良い.

(ii) ヒープ方式とスパゲティ方式の優劣は微妙で, 総合効率ではヒープ方式が優るが, 環境の保持が起こらないようなプログラム及びページング環境では, 明らかにスパゲティ方式の方が良い.

(iii) スタック・ヒープ併用法とスパゲティ方式では平均 30% 併用法の方が効率が良い. また, 併用法はヒープ方式に対しても平均 10% 効率が上である.

これら2つの性能評価は, いずれもある効果的な解答を示してはいるが, 基準が妥当かどうかという問題点もあり, 万能とは言い難い. どこに妥協点を見い出すかは, 処理系作成者の意志にまかされている, というのが現状なのである.

5. プログラミングと多重環境

これまで述べてきたような, 強力で一般的なモデルでは, 効率低下は避けられず, またプログラミングの際の記述も繁雑になりがちである. しかし, コルーチンの考え方は, プログラムの自然な分割を可能にし, 分割された各モジュールの独立性を高めるなど, プログラム作成に, 極めて有効な概念である. そこで, この有効性と, 現実性を持ったセミコルーチンに的を絞ってみる.

一つの主コルーチンが, 複数の従コルーチンを呼ぶ場合に, 従コルーチンが呼び出された順に長い寿命を持つという仮定をおけば, これらは, 完全にLIFOのスタックで実現できる. さらに, 従コルーチンが主コルーチンに値を渡す作業の繰り返しとして抽象化すれば, CLU や Alphard のイテレータの機構に到達する. (このイテレータは, 簡単に実現できる上に, 制御の抽象化が効果的に行え, また手続き呼び出しなどで, 同様の作業を記述した場合の1/4の制御管理で済むという報告[17]もある.) コルーチンでは重すぎるとするならば, セミコルーチンやイテレータなどは, 現実的な選択枝の一つであると言えるだろう.

6. おわりに

スパゲティ・スタックモデルの論文が出て 10 年が

* 主従関係が定まっているコルーチン. 主コルーチンは従コルーチンが終る前に終了してしまうことはない.

たち，その姿は，除々に実用のレベルに近づきつつあるが，プログラミングの際の記述法は，洗練されてきたとはとても言えないのが現実である．元来，環境の保持を考えない手続きを基本に設計されてきた言語仕様には，相容れないものかもしれない．根本的な発想の転換が必要かもしれない．いずれにしても，これからは，多重の環境が自然に記述できるような，新しいプログラミング言語の研究が待たれるところである．

参 考 文 献

1) Berry, D. et al.: Time required for reference count management in retention block-structured languages, part 1, Int. J. Comput. Inform. Sci., Vol. 7, No. 1, pp. 11-64 (1978).
2) Berry, D. and Sorkin, A.: Time required for garbage collection in retention block-structured languages, Int. J. Comput. Inform. Sci., Vol. 7, No. 4, pp. 361-404 (1978).
3) Bobrow, D. and Wegbreit, B.: A Model and Stack Implementation of Multiple Environments, Comm. ACM, Vol. 16, No. 10, pp. 591-603 (1973).
4) Dahl, O.-J., Myhrhaug, B. and Mygaard, K.: Simula 67 Common Base Language, Norwegian Computing Center Publication No. S-2 (1968).
5) Greenblatt, R.: The LISP Machine, MIT AI Lab. Working Paper 79 (Nov. 1974).
6) Hanson, D. R. and Griswold, R. E.: The SL 5 Procedure Mechanism, Comm. ACM, Vol. 21, No. 5, pp. 392-400 (1978).
7) Kearns, J. P., Meier, C. J. and Soffa, M. L.: The Performance Evaluation of Control Implementations, IEEE Trans. Softw. Eng., Vol. SE-8, No. 2, pp. 89-96 (1982).
8) Marlin, C. D.: Coroutines, Lect. Notes in Comptr. Sci., Vol. 95, Springer, Verlag (1980).
9) Mc Dermott, D. V. and Sussman, G. J.: The Conniver Reference Manual, MIT AI Memo No. 259a (1974).
10) Pauli, W. and Soffa, M. L.: Coroutine behaviour and implementation, Softw. Pract. Exper. Vol. 10, No. 3, pp. 189-204 (1980).
11) Prenner, C. J., Spitzen, J. M. and Wegbreit, B.: An Implementation of Backtracking for Programming Languages, Proc. 27th Nat, Conf, ACM, pp. 763-771 (1972).
12) Reynolds, J. C.: GEDANKEN-A Simple Typelees language. based on the principle of completeness and the reference concept, Comm, ACM, Vol. 13, No. 5, pp. 308-319 (1970).
13) Steele, G. L.: Macaroni is better than Spaghetti, Proc. of the Symp. in AI and Prog. Lang., SIGPLAN Notices, Vol. 12, No. 8, pp. 60-66 (1977).
14) Teitelman, et al.: INTERLISP Reference MANUAL, Xerox Palo Alto Research Center (1975).
15) Weinreb, D. and Moon, D.: Lisp Machine Manual, Fourth Edition (July 1981).
16) 安部憲広：コルーチンとは, bit, Vol. 12, No. 2, pp. 35-40 (1980).
17) 佐渡一広：プログラム言語 CLU の実用的処理系と，その使用経験，情報処理学会論文誌，Vol. 22, No. 4, pp. 295-303 (1981).
18) 島田俊夫，山口喜教，坂村 健：LISP マシンとその評価，電子通信学会論文誌，Vol. J59-D No. 6, pp. 406-413. (1976).
19) 東出正裕，小西啓二，安部憲広，辻 三郎：B フレーム・Mフレームを使用したミニコンリスプ FLISP. 情報処理学会論文誌，Vol. 20, No. 1, pp. 8-16 (1979).

(昭和 57 年 12 月 7 日受付)