25 周年記念論文

「多態実行環境」: 高級言語の制御機械の高性能実現法†

鈴木則久††　小方一郎†††

高機能ワークステーションは人工知能，プログラム作製，CAD などに使われる．最近の一つの傾向として全システムが単一言語で記述してある．LISP マシン，Smalltalk マシン，Prolog マシンなどがこれである．このように全システムを単一言語で記述すると開かれたシステムとなり，ユーザが簡単にカスタム化できる CAD システムなどが可能となる．しかし，このように全システムを単一言語で記述するためには言語がオペレーティング・システムを記述する機能を持っていなければならない．コルーチン，プロセスとか，実行環境をデータとして扱ってデバッガを作る機能である．これらを言語に取り入れる手段として実行環境を複雑に扱わざるをえない．InterLisp ではスパゲッティ・スタック，LispMachine Lisp ではスタック・グループを導入した．一方 Smalltalk-80* では実行環境をオブジェクトとして扱っている．これは強力であるが実行速度には問題があった．筆者たちは，高速実行のときは線型スタックに実行環境を作り，複雑な制御をするときはヒープに実行環境をとる，多態実行環境というデータ構造を考案し，Smalltalk-80 の高度柔軟性を保ちながら，高速に実行できるシステム「菊 32V」を開発した．これは現在，32 ビットのインタプリタとしては世界最高速である．

1. まえがき

高機能ワークステーションの重要な用途は人工知能，プログラム作製，CAD である．最近の高機能ワークステーションの傾向の一つとして，単一言語で全システムを記述してある．LISP マシン，UNIX マシン，Smalltalk マシン，Prolog マシンなどである．単一言語で全システムを記述すると，システムあるいはアプリケーション・プログラムを簡単にカスタム化できる開いたシステムができる．しかしながら，そのためには言語にプロセス，記憶管理，実行環境アクセスなどオペレーティング・システムの機能がなければならない．

また単一言語で全システムを書くと移植が容易になる．その上，ユーザがプログラムを開発する能率もあがる．これは，システムで使っているプログラムをそのままユーザのプログラムから使えるからである．

しかしながらオペレーティング・システムの機能を高級言語の中に取り入れようとすると問題になるのは制御である．PASCAL のように再帰呼び出しのある言語では，プロシージャの呼び出しが完全入れ子構造になっているので，スタックで実現できる．しかし，オペレーティング・システムではプロセス・スイッチやコルーチンなどがあり，LIFO スタックでは実行環境の管理ができない．

そこで最近の高級言語では複雑な実行時制御用データ構造をそなえている．InterLisp ではスパゲッティ・スタックを持っており，MacLisp ではスタック・グループを持っている．

一方 Smalltalk では[1] 実行環境（**コンテクスト**と呼ばれる）をオブジェクトとしてヒープに割り付け，消滅はガーベッジ・コレクタが行う．**ファンアルグ問題**も解決され，強力な言語となっているが，実行速度の速いシステムを作るのは大変困難である．

われわれはこれらのシステムを能率良く実現する方法として**多態実行環境**というものを考案した．実行環境として実行速度の高い形（**揮発性コンテクスト**と呼ぶ）と，データとして扱いやすく，またコンテクストの保持（retention）を正しく実現している形（**永久コンテクスト**と呼ぶ）の二つを持ち，実行状況に応じて，両者の間で変態する．統計的に見ても，実行環境をオブジェクトとして扱わなければならないのはまれで，大体の場合はメソッドの呼び出しと帰還は入れ子構造になっている．われわれはこのアルゴリズムを使って Smalltalk-80 の仮想機械「菊 32V」を作製した．これは 32 ビットのインタプリタとしては世界最高速で

† Polymorphic Context: an Efficient Implementation Technique for High-Level Language Control by Norihisa SUZUKI (Mathematical Engineering Department, University of Tokyo) and Ichiro OGATA (Electro Technical Laboratory).

†† 東京大学工学部計数工学科
††† 電子技術総合研究所

* Smalltalk-80 はゼロックス社の登録商標である．

ある.

Smalltalk の仮想機械は1970年代の初頭からゼロックス・パロ・アルト研究所で作られてきたが，長い間実行環境はヒープに取られてきた．この方法で作られた仮想機械には Dorado Smalltalk と Dolphin Smalltalk があり，前者は毎秒 25 万バイトコード，後者は毎秒2万5千バイトコード実行できる．Dorado はマイクロプログラムの実行では 4MIPS の高性能パーソナル・コンピュータである．その後他の会社にもライセンスがおりシステム作成の研究が進められたが，どれも Dolphin Smalltalk を越える速度のものにはならなかった[2]．ところが1984年になって，種々のグループで独立にこの論文で述べる方法と同様にコンテクストを線型に割り付ける技術が開発され，Deutsch & Schiffman のシステム[3]，Tektronix 4404, BSII,「菊 32V」というどれも MC 68000 マイクロプロセッサ上で動く高速なシステムが開発された．

Smalltalk あるいはそれと同様に複雑な制御構造を持つ InterLisp や LispMachine Lisp が標準マイクロプロセッサ上で，マイクロコードのできる専用マシンよりも高速に実現できたことの意味は大きい．まず，MC 68020 が出荷されはじめているが，これは MC 68000 の4倍の速度が出ると言われており，Deutsch & Schiffman の Smalltalk が 68020 の上で動くと Symbolics 3600 上の Zeta Lisp と同じ速度になる．また，低価格の方では MacIntosh のようなパソコン上でも高速に動くシステムができうる．

Symbolics 3600 上では開かれた CAD システムがすでに実用になっているが，同様なことが Smalltalk をはじめとするその他の高級言語でも，もっと低価格で実現するであろう．

2. Smalltalk-80 仮想機械の実行環境と問題点

2.1 Smalltalk-80 の実行環境

Smalltalk-80 は共有変数で通信する多重プロセスによる並列性や，関数パラメタを持ち，その変数のバインディングは静的スコープである．よってその実行環境は最も複雑な LISP (たとえば InterLisp) のそれとほぼ同じである．

一方，Smalltalk-80 に特別な機能は，その実行環境をオブジェクトとしていることである．したがって，実行環境は常にデータとして扱うことができる．これによるメリットは数多い．

第一に，マルチ・プログラミング・システムの記述が Smalltalk 自身でできる．実際システムはそのように構成されており，ユーザはプロセスの生成・消去，スケジューリング，同期などもプログラムで記述することができる．

第二に，デバッガの簡明な実現ができる．あるプロセスの実行環境を，デバッガのプロセスからオブジェクトとして扱い，読んでまた書き換えることにより，非常に柔軟なデバッギングが可能となる．

第三に，いわゆる**環境問題** (funarg problem) の解決が可能となる．**環境問題**には**下方環境問題** (downward funarg problem) と**上方環境問題** (upward funarg problem) とがある．下方環境問題は，静的スコープを持つ言語で関数引数も許すとき，ある関数 f が，それを定義している関数 g 内の一時変数を参照しており，かつそれが他の関数呼び出しの引数として渡されたときにも f が呼ばれたときには正しく g 内の一時変数を参照できることである．上方環境問題は，関数 f が関数 g の結果として返されても，f が呼ばれたときはもうすでに終了している g の一時変数を参照できることをいう．

上方環境問題，下方環境問題を正しく扱う機能は，関数をもオブジェクトとして扱える近代的なプログラミング言語では備えられるべき機能である．InterLisp ではスパゲッティ・スタック[4]を使って同様の機能を実現している．しかし，この実現は複雑で正しい製作には多くの日時を要した．Smalltalk-80 仮想機械のように「実行環境をオブジェクトとする」実現方法は，簡潔であり他のシステムの作製も容易になり非常に魅力的である．

しかし，後に述べるように，この方法は実行効率上で問題を持っている．

そこで，効率を改善し，かつ記述上の簡潔さをも追及し，実行環境を，

1. 実行に適した形態
2. オブジェクトの形態

の二種の形態で表し，必要に応じて相互に移り変われるようにしたものが，この論文で述べる「多態実行環境」である．われわれは実際にこのアルゴリズムを使ってシステムを作製し，実行環境のプログラム上での簡潔な取り扱いと，実行効率が十分に両立することを示すことができた．

われわれは Smalltalk-80 の仮想機械を多態実行環境を使って実現した．これは「**菊 32V**」と呼ばれるマ

イクロプロセッサ MC 68000 用のシステムで，C で記述されている．これは，Cで書かれた Smalltalk-80 の仮想機械としては世界で最も速く，UC バークレー (カリフォルニア大学バークレー) で作られた **BSII**[5] より2割，ゼロックス社の **Dolphin** より 3 割程速い．また「菊 32V」は PC 9801 にも移植され[6]ており，Dolphinと 同程度の速度で動く．

また，多態実行環境は Smalltalk-80 でのみ有効なアルゴリズムではない．言語の中でさまざまなオペレーティング・システムの機能，すなわちプロセス管理，デバッガなどを記述できるプログラミング言語の実現に応用できる．その意味では InterLisp のスパゲッティ・スタックの代わりに使うこともできる．スパゲッティ・スタックと多態実行環境のどちらが能率が良いかは興味ある研究課題である．

さらに，ここで取り上げた基本的哲学である，記述に簡潔であるデータ構造と，実行効率の良いデータ構造を両方実行時に持たせ，状況に応じて相互に変換して使うという考えは，広く計算機科学の他の分野にも応用できる有効な手法である．

2.2 変数環境

多態実行環境の具体的なアルゴリズムの説明にはいる前に，Smalltalk-80 仮想機械のデータ構造を見てみよう．

Smalltalk-80 は七種類の変数を持っている．すなわち，

1) グローバル変数：あらゆるクラスのメソッドからアクセスできる．

2) クラス変数：一つのクラスのメソッドからアクセスできる．

3) プール変数：プールを含むと宣言したクラスのメソッドからアクセスできる．

4) 擬変数 (self, super など)：代入ができない変数．

5) インスタンス変数：インスタンス固有の変数．

6) 一時変数：メソッド固有の変数で，メソッドが呼ばれたときのみ値が有効．

7) 引数：メソッドのパラメータ．

これらの変数は，Smalltalk-80 のセルフ・コンパイラでコンパイルされ，バイトコードとリテラルで表現される．このバイトコードはインタプリタで解釈・実行される．

一方インタプリタ側から見た場合は，変数はその格納場所により以下の三種にコンパイルされている．

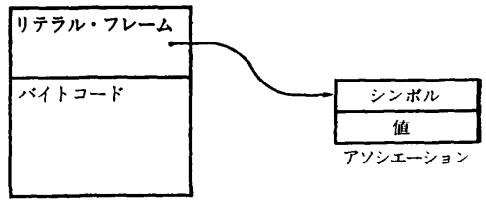

図-1 リテラル変数
Fig. 1 Literal variable.

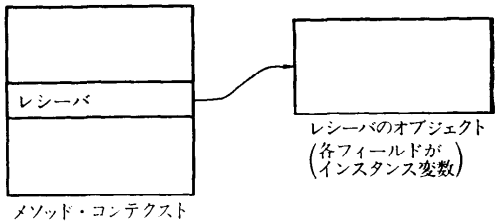

図-2 インスタンス変数
Fig. 2 Instance variable.

a) リテラル変数

b) インスタンス変数

c) テンポラリ変数

a)のリテラル変数は，メソッドのリテラル・フレームから参照された，静的に割り当てられた変数であり，グローバル，クラス，プールの各変数がこれにあたる (図-1)．

b)のインスタンス変数は，実はメソッドを起動したレシーバの内部変数である．5)のインスタンス変数が相当する (図-2)．

最後に，c)のテンポラリ変数だが，これは，メソッドの起動ごとに動的に割り当てられる変数である．これは，メソッドの再帰呼び出しを保証するためである．6),7)の一時変数及び引数がこれに相当している．4)の擬変数はa)かc)である．

このように Smalltalk-80 の変数環境は，変数が完全に静的にバインドされている上，すべてのメソッドが独立であるので，非常に簡明となっている．

2.3 実行環境

Smalltalk-80 では普通のプログラミング言語の手続きに相当するものをメソッドと呼ぶ．また手続きをパラメタとして渡したりするために，すぐには評価しないプログラムのかたまりをブロックと呼ぶ．

メソッドとブロックの実行を実現するために Smalltalk-80 仮想機械には，メソッドを起動するたびに作られる**メソッド・コンテクスト**と，ブロック式 (block) に出会うたびに作られる**ブロック・コンテクスト**がある．

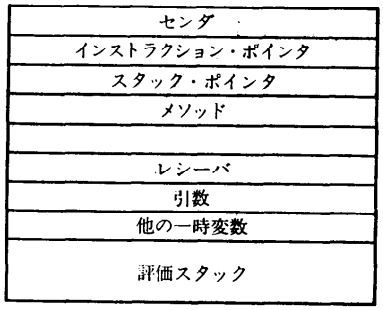

図-3 メソッド・コンテクスト
Fig. 3 Method context.

Smalltalk-80 バイトコードはスタック操作が主体となっている．したがって，二項演算の引数も含め，あらゆるメッセージ・センドの引数は**評価スタック**上に置かれる．各メソッドごとに必要な評価スタックの深さはコンパイル時に判るので，これをメソッド・コンテクストの中に割り当て，メソッドの起動ごとに新しい評価スタックを使えるようにしなければならない．

また，各メソッドの引数と一時変数はメソッド・コンテクストに取られる．

あるメソッドが他のメソッドを起動すれば，もとのメソッドの実行はサスペンドされる．このとき，再開に備えて，バイトコード・インストラクション・ポインタと評価スタックへのスタック・ポインタは待避しなければならない．

これらの，メソッド実行に必要な記憶領域を持つオブジェクトがメソッド・コンテクストである（**図-3**）．

一方ブロックはメソッド内で定義されており，ブロックを実行するためのバイトコードは定義されたメソッドをコンパイルしてできるコンパイルド・メソッド内に，また，ブロックからアクセスできる一時変数はブロックの定義されたメソッド・コンテクスト（**ホーム・コンテクスト**と呼ぶ）内に置かれる．また，引数付きのブロックではその引数を配置する場所が必要だが，これもホーム・コンテクストに取られる．ブロックが入れ子になっていて，それぞれに引数がある場合も，記憶域はホーム・コンテクストがあてられる．

これらの情報を持っているのがブロック・コンテクストである．この構造は**図-4** に示される．

2.4 効率上の問題点

Smalltalk システムを，仮想機械の仕様で単純に実現すれば，メソッド起動のたびにコンテクストが一個生成される．しかし，これでは次の理由からメソッドの起動が遅くなる．

まず第一に，実行環境用のオブジェクトを記憶システムから割り当て，各フィールドには nil を代入して初期化する必要がある．メソッド・コンテクストは比較的大きいオブジェクト（20語と40語）であるため，初期化の手間も大きい．

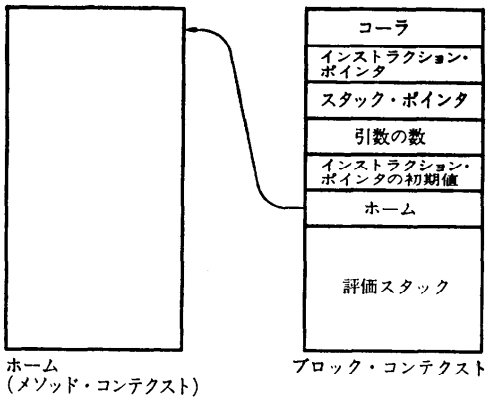

図-4 ブロック・コンテクスト
Fig. 4 Block context.

第二にデータ変換の手間がある．現在実行中のコンテクストをアクティブ・コンテクストと呼ぶが，インタプリタはインストラクション・ポインタやアクティブ・コンテクスト上のスタック・ポインタを絶対アドレスで持っている．他のメソッドを起動する際には，これを各オブジェクトの先頭からの距離という相対アドレスに変換し，さらに Smalltalk の整数に変換してアクティブ・コンテクストの中に待避し，それからそのコンテクストをサスペンドする．なぜこのようなデータ変換をしなければならないかというと，ガーベッジ・コレクションのときポインタと区別しなければならないから，ポインタでないときは整数に変える．またコンパクションをすると，オブジェクトが動くので，相対番地で持っていなければならない．

第三に引数の受け渡しの手間がある．メソッドが起動された場合，引数は呼び出したコンテクストの評価スタックから，呼び出された側のコンテクストの引数のフィールドへコピーする必要がある．

第四に，メソッド探索の手間がある．メッセージ・センドがおこると，メソッドが呼び出されなければならないが，メッセージ・センドが規定しているのは，メッセージ・セレクタだけである．同じセレクタを持つメソッドはいろいろなクラスで定義することができ，実際にどれを呼び出すかは，レシーバのクラスによる．

これが Smalltalk 仮想機械でのメッセージ・センドのときに起こるボトルネックであり，1984年以前

に実現されたシステムでは皆存在する.

2.5 メソッドの実行

加えて, Smalltalkは徹底したオブジェクト指向言語であり, **PASCAL** のような手続き型言語や, **Simula** のようなオブジェクト指向言語に比べてもメッセージ・センドの頻度が高い.

この理由としては, 第一にオブジェクト指向言語では, 情報隠蔽の機能があるが, これを実現するため, オブジェクトへのアクセスはすべてメソッドの呼び出しを伴うからである.

第二に Smalltalk では算術の演算や配列のアクセスまでメソッドの呼び出しで実現しているからである. こうする利点は, プログラムの共用性が高くなり, 再利用がしやすくなるからである.

このような性質のため, Smalltalk においては, メソッド起動の頻度は高く, その速度がシステム全体の速度に重要な影響を与える.

3. 多態実行環境

Smalltalk 仮想機械のインタプリタの仕様を完全に満たし, 前述のメッセージ・センドに伴うコンテクストの問題を解決するのが多態実行環境である.

多態実行環境とは, コンテクストを

1) 実行に適した, リニア・スタックの形

2) データとして扱えるオブジェクトの形

の二通りの表現を取れるようにし, 場合場合に応じて形態を変えるコンテクストを言う. リニア・スタックの形は PASCAL の実行環境と同一であり, 実行効率が高いが, Smalltalk の意味に適合した実行には完全には対応できない. 一方オブジェクトの形は, Smalltalk 仮想機械の仕様に等しい.

当然予想されるように, コンテクストをデータとして扱うのは, 頻度の少ない操作である. コンテクストをオブジェクトとして扱わなければならないのは, プログラムがコンテクストを“参照”した場合に限られる.

普通に Smalltalk のプログラムを実行しているときは, ほとんどのコンテクストはプログラムから参照されることはなく, 生成されてすぐ消滅するであろう. こういう形で実行されているかぎり, コンテクストは生成されたのとは逆の順に消滅されていく. よってコンテクストは LIFO スタックに割り付けることができる.

しかしながら, コンテクストが参照されると, LIFO スタック上のコンテクストはすべてオブジェクトに変換される.

Smalltalk のインタプリタが, コンテクストを参照するのは, 次の三つの場合に限られる.

1) 割り込みにより, プロセス・スイッチが起きる場合.

2) 特殊なバイトコード “push active context” を実行した場合. (主に遅延評価するブロック式に出合った場合に実行される.)

3) プリミティブである primitive value (ブロック・コンテクストを実行するプリミティブ) を実行した場合.

3.1 多態実行環境の実現法

コンテクストはリニア・スタックに割り付けられる形とオブジェクトの形があるが, 前者を揮発性コンテクスト, 後者を永久コンテクストと呼ぶ.

コンテクストはメソッドが起動されるたびに, 最初は揮発性コンテクストがリニア・スタック上に割り付けられる. 揮発性コンテクストへの参照が起こらないかぎり, 揮発性コンテクストはリニア・スタックに沿って次々と割り付けられ, また解放されていく. 揮発性コンテクストはオブジェクトではないので, ヒープへもオブジェクト・テーブルへも割り付けは起こらない. 参照がおこると揮発性コンテクストは永久コンテクストになる. 揮発性コンテクストはプログラムからアクセスされないから, インストラクション・ポインタはデータ変換の必要はなく, 揮発性コンテクスト内に絶対番地のまま格納される. 揮発性コンテクストを, 永久コンテクストに変換する際にはこれらの情報も同時に変換されなければならない.

またリニア・スタック上では呼び出す側のコンテクストのすぐ隣りに呼び出される側のコンテクストがくるので, 引数は両者で共有でき, コピーの必要はなくなる.

ヒープにコンテクストをオブジェクトとして作っていたときは, 全フィールドを nil で初期化しなければならなかったが, 揮発性コンテクストでは引数以外の一時変数だけ初期化すればよいので, ずっと手間がはぶける.

これらの要求を考慮して考えられた揮発性コンテクストは**図-5** のような形をしている.

リニア・スタックは実際にはあまり大きく取る必要がない. われわれの作製した「菊 32V」では 256 語 (4バイト／語)取ってあるが, 128 語でも標準ベンチ

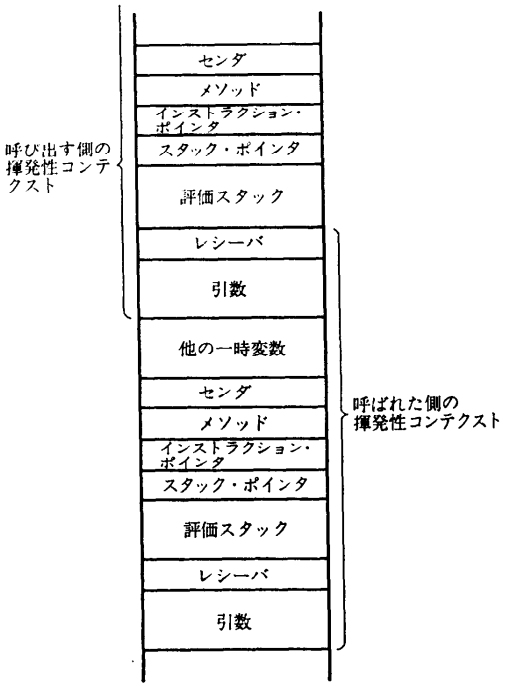

図-5 揮発性コンテクスト
Fig. 5 Volatile context.

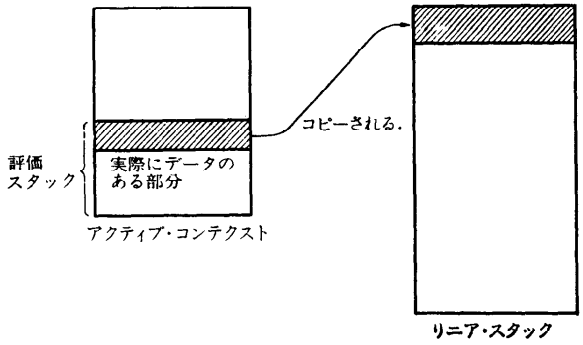

図-6 リニア・スタックの底には，永久コンテクストの試価スタックの部分がコピーされる
Fig. 6 Copying evaluation stack of eternal context to the bottom of linear stack.

マークを走らせているかぎりでは性能の変化は全く認められなかった.

しかし，リニア・スタックは有限なので，**オーバフロー**した場合，**アンダフロー**した場合の処理を考えなければならない.

オーバフローしたときは，すべての揮発性コンテクストを永久コンテクストにして，リニア・スタックを空にする．またアンダフローした場合には，底にあったコンテクストのセンダ・フィールドが指している永久コンテクストをアクティブ・コンテクストにする.

ここで詳しく説明しなければならないのが，リニア・スタックの底にあるコンテクストの形態である．われわれのインプリメンテーションでは，底には必ず永久コンテクストの評価スタックだけを置く.

システムがスタートされたときには，バーチャル・イメージの中からアクティブ・コンテクストが読み出される．この中の評価スタックの部分をリニア・スタックにコピーし，制御はアクティブ・コンテクストに渡される（**図-6**）.

この後ロードとかストア命令が起こっても，オブジェクト・ポインタはリニア・スタックにしかつまれず，アクティブ・コンテクストの評価スタックの中味は陳腐化する.

ここでメッセージ・セントが起こると，揮発性コンテクストが，この上につまれていく.

また，コンテクストへの参照が起こると，揮発性コンテクストは上から順々に永久コンテクストに変換されていき，リニア・スタックからポップされていくのだが，最後にくるとそれは永久スタックの評価スタックの部分なので，元の永久スタックにコピー・バックして，コンテクスト・スイッチの前半は終わる．それから，次にアクティブになる永久コンテクストが選ばれて，図-6 のように評価スタックがコピーし直される.

3.2 ガーベッジ・コレクションとの関係

最近の高級言語では皆自動的に記憶域の回収を行っており，Smalltalk でも同様である．ただ Smalltalk で違うことはインタアクティブ・システムであるので，**実時間ガーベッジ・コレクション**をやらなければならないことだ．「菊32V」では**ドイッチュ・ボブロー遅延ガーベッジ・コレクション**[7] を使っている．この方法での利点は，作られてからすぐなくなるオブジェクトにはレファレンス・カウンティングはしないことだが，多態実行環境ではリニア・スタックの中味がまさしくこのレファレンス・カウンティングをしないオブジェクト・ポインタとすることができる.

3.3 多態実行環境の評価

多態実行環境の利点の第一は，揮発性コンテクストの形と内部データの形は，オブジェクトである永久

コンテクストに変換するのに十分な情報が格納されていれば，どのような形でもよい．この形では決してSmalltalk バーチャル・イメージ側からアクセスは起こらないからである．実行に適するよう，インプリメントする計算機の機械語と整合性がよく，高速に実行できるような形に決めることができる．

第二にこの形ではコンテクストは一定の場所に割り当てられるため，記憶参照の局所性が増す．ページ単位でスワップする仮想記憶システムでは，ワーキングセットを小さくし，また主記憶に常駐させる領域も小さくして効率を上げることができる．

第三に，ガーベッジ・コレクションとの整合性がよい．このことはすでに述べたが，ドイッチュ・ボブローの遅延レファレンス・カウンティング・ガーベッジ・コレクションでは，リニア・スタックをレファレンス・カウントをしない場所とすることができる．

3.4 多態実行環境の問題点と改良案

Smalltalk ではブロックは例外処理として使われることが多い．したがってこのブロックは生成されても評価されることは少ない．

またもう一つのブロックの用途は制御構造の実現である．たとえば反復文(for 文)に対応する，to:do:メッセージがそれである．この場合は，逆に同じブロックが何度も評価されることになる．また，これは言語の意味の定義の問題だが，反復文が入れ子になっていると，内側のブロックは反復のたびに生成されなければならず能率が落ちる．こういう問題点から，Smalltalk-80 のコンパイラでは，to: do: と if True: if False: は特別扱いして，メッセージ・センドなしのバイトコードを直接生成している．

さて，われわれの考案した多態実行環境ではブロック・コンテクストが生成されるときは，揮発性コンテクストから永久コンテクストへの変換が起こり，線形スタック上のコンテクストは全部ヒープに移される（これをスタックをフラッシュ・アウトすると言う）．ブロック・コンテクストが評価されるときも (value が送られたとき) スタックはフラッシュ・アウトされる．

よって前者のように例外処理をしようとする場合には，例外が起こらない場合(大部分がそうである) でもスタックをフラッシュ・アウトせねばならず無駄である．また後者の場合には評価が何回も起きてそのたびにスタックをフラッシュ・アウトせねばならない．

前者のようにブロック・コンテクストを作る場合でもスタックをフラッシュ・アウトしないですませるのはむずかしい問題である．ブロック・コンテクストを生成するときは，アクティブ・コンテクストへのポインタを作る．それでアクティブ・コンテクストをオブジェクトとして扱う可能性も出てくるので，スタックをフラッシュ・アウトしなければならない．しかし，例外処理の場合はこのブロック・コンテクストへのポインタはさらにリニア・スタックの後の方に作られるメソッド・コンテクストの一時変数として格納されるだけでリニア・スタックの外へ持ち出されることはない．そこでブロック・コンテクストの指すアクティブ・コンテクストはブロック・コンテクストのポインタより必ずリニア・スタックの深い方にあり，ブロック・コンテクストへのポインタがリニア・スタックの外へ持ち出されないなら，スタックのフラッシュ・アウトはブロック・コンテクストへメッセージが送られたときだけでよい．しかし，インタプリタでポインタの持ち出しを検出しようとすると，オーバヘッドが多くなって実用的でない．コンパイラを作成して，チェックすればかなりの場合はスタックをフラッシュ・アウトしなくてすむであろう．

しかし，ブロック・コンテクストを評価するときにスタックをフラッシュ・アウトするのをやめるよう，インタプリタで検出するアルゴリズムはオーバヘッドが小さく，必ずフラッシュ・アウトをやめることができる．まず，ブロック・コンテクストの評価が起これば，その評価スタックはリニア・スタック上に取る．また，一時変数領域はホーム・コンテクストにあるが，これはブロック・コンテクストから参照されてい

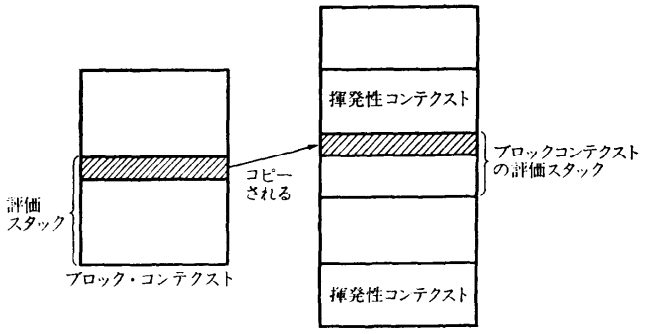

図-7 ブロック・コンテクストを評価するときのリニア・スタックの取り扱い
Fig. 7 Maintenance of linear stack during evaluation of block context.

るので，永久コンテクストである．よって一時変数はこの永久コンテクストに格納する．リターン文を実行して，スタックをポップするときは，ブロック・コンテクストの評価スタックには特別な印があり，単にフラッシュ・アウトするのではなく，ブロック・コンテクストの評価スタックにコピーしなければならない(図-7)．

4. 性能評価

Smalltalk には，システムの性能を評価するためのテストプログラムを50個程と，その計時およびフォーマット付き印刷をするためのメソッドを含んだクラス Benchmark があり，Smalltalk システムはこれを動かして性能評価し，他のシステムと比べられる[2]．

Benchmark には二種類あり，マイクロベンチマークという個々のバイトコードあるいは Smalltalk の文の性能を評価するテストプログラムを持ったものと，マクロベンチマークというかなり大きな典型的なプログラムとがある．

いくつかのマイクロベンチマークのテストプログラムとマクロベンチマークのすべてのプログラムについて数機種の性能を比較したのが表-1である[2]．

ここで比較しているのは

a) Dorado[8]：ゼロックス PARC で開発された ECL を使用した高性能ワークステーション．

b) Kiku 32V：筆者たちが SUN 2 (MC 68010) の

表-1 Smalltalk システムの性能比較
Table 1 Benchmarks of smalltak systems.

各ますの中の数値は，Dorado を100とした時の速度（大きい方が速い）である．
(Dorado=100)

マイクロベンチマーク	Kiku 32V	BS II	Lisa	Dolphin
1. load an instance variable	6	6	2	6
2. store into an instance variable	3	4	2	6
3. divide 3 by 4	62	57	44	14
4. activate and return	15	12	4	9
5. send #at:	12	15	3	8
6. execute blockcopy:	8	5	3	3
7. call bit BLT	21	17	8	17
8. scan characters (primitive text display)	46	27	32	19
マクロベンチマーク				
9. read and write class organization	12	11	5	10
10. print a class definition	12	11	5	10
11. print a class hierarchy	16	11	7	12
12. find all calls on #print String Radix:	10	9	5	8
13. find all implementors of #next	11	10	5	10
14. create an inspectors view	14	14	6	12
15. compile dummy method	13	11	4	11
16. decompile class Input Sensor	12	9	6	11
17. text keyboard response using lookahead buffer	16	17	9	15
18. text keyboard response for single keystroke	14	15	8	15
19. display text	18	13	10	13
20. format a bunch of text	17	17	6	13
21. text replacement and redisplay	23	21	11	16
平均（7～21までの算術平均）	17	14	8	13

上で開発したシステム.

c) BSII[9]: カリフォルニア大学バークレーで開発された SUN 2 上のシステム.

d) Lisa: アップル社で Lisa 上に開発したシステム.

e) Dolphin: ゼロックス PARC で開発された TTL を使用したワークステーション (1100 SIP).

またこの表の数値は Dorado での速度を 100 としたときとの比でもって表している. すなわち表の値が 10 のときは Dorado の十分の一の速度ということである.

たとえば, コンパイラの速度はベンチマーク 15 で比較される. 表-1 によると, Kiku 32V では Dorado の 13%, Dolphin では 11% である.

しかしこの表を見ても, システムにより得手不得手があり比較がむずかしい. そこで全体の性能の目安としてゼロックス PARC では 7 から 21 までの数の算術平均を使うことを推めている. それが, 平均という値である. Kiku 32V は Dorado の六分の一で Dolphin より三割速い.

5. むすび

われわれは三年間にわたりオブジェクト指向言語のシステムを作る研究を進めてきた[10),11)]. この間には Smalltalk のシステム四つを含め七つのシステムを作り, 種々の実現法について検討を重ねた. ここで発表した多態実行環境はこの連続した研究の中から生まれたアルゴリズムであり, 実際にそれを使ったシステム「菊 32V」を作り, 32 ビット・オブジェクト・ポインタを持つ Smalltalk-80 のインタプリタとしては世界最高速のものを実現し, また市販されているゼロックス 1100 SIP よりも三割速いことを実証した.

また, 実行環境のさらに効率良いアルゴリズムとして, ブロック・コンテクストを評価してもスタックをフラッシュしなくてもすむアルゴリズムを示した.

実際にこの多態実行環境のアルゴリズムを使ってマイクプロセッサ「刀 32」を設計し, 現在製作中である[12)].

「菊 32V」をはじめ多くのシステムのグラフィックスと入出力プログラムは寺田実氏が作製した.

参考文献

1) Goldberg, A. and Robson, D.: *Smalltalk-80: The Language and Its Implementation*, Addion-Wesley, Reading, MA (1983).
2) Krasner, G. ed.: *Smalltalk-80: Bits of History, Words of Advice*, Addison-Wesley, Reading, MA (1983).
3) Deutsch, L. P. and Schiffman, A. M.: Efficient Implementation of the Smalltalk-80 System, *Proc. of the 11th Ann. ACM Symposium on the Princ. of Prog. Lang.*, ACM, New York (1984).
4) Bobrow, D. G. and Wegbreit, B.: A Model and Stack Implementation for Multiple Environments, *Comm. ACM*, Vol. 16, No. 10, pp. 591-603 (1973).
5) Unger, D. M.: Generation Scavenging: A Nondisruptive High Performance Storage Reclamation Algorithm, *Proc. ACM Soft. Eng. Symp. on Pract. Soft. Devel. Env.*, ACM, New York (1984).
6) 鈴木則久, 小原盛幹, 中島 淳: PC-Smalltalk, 情報処理学会オペレーティング・システム研究会資料, No. 27 (1985).
7) Deutsch, L. P. and Bobrow, D. G.: An Efficient, Incremental, Automatic Garbage Collector, *Comm. ACM*, Vol. 19, No. 9, pp. 522-526 (1976).
8) Lampson, B. W. et al.: *The Dorado: A High Performance Personal Computer*, Xerox PARC Technical Report CSL-81-1 (Jan. 1981).
9) Smalltalk-80 Newsletter, Xerox PARC, No. 4 (Sept. 1984).
10) Suzuki, N. and Terada, M.: Creating Efficient Systems for Object-Oriented Languages, *Proc. of 11th Ann. ACM Symp. on Princ. of Prog. Lang.*, ACM, New York (1984).
11) Suzuki, N., Ogata, I. and Terada, M.: *Design of a 32-bit Virtual Machine of Smalltalk-80*, 情報処理学会記号処理研究会資料, 84-28 (1984).
12) Suzuki, N., Kubota, K. and Aoki, T.: Sword 32: a Bytecode Emulating Microprocessor for Object-Oriented Languages, *Proc. of FGCS 84*, ICOT (1984).

(昭和 60 年 7 月 1 日受付)