3N－2

ISLisp言語処理系のバイトコードインタプリタの実装

船戸潤一　前田敦司　中西正和

慶応義塾大学理工学部数理科学科

1. はじめに

Lisp は 1962 年の Lisp1.5 の発表以来、記号処理用言語として発展し続けている。そのなかで、ISO(International Organization for Standardization) により設計された Lisp の標準化案に、ISLisp[1] がある。ISLisp は、Common Lisp をベースの言語とし、小さく、効率の良い処理系の作成を目的として設計された Lisp 言語である。

本研究は、ISLisp 言語処理系の実装を目的とする。本システムは原始プログラムを中間コードに変換するコンパイラ部分と、中間コードを解釈、実行するインタプリタ部分とから成り、中間コードにはバイトコードを用いる。本稿では ISLisp 言語処理系のインタプリタ部分であるバイトコードインタプリタ ILBI について、その実装方法と、インタプリタの効率化について報告する。

2. バイトコード

バイトコードは中間言語の一つであり、各命令が 1 バイトから成るために、このように呼ばれる。バイトコードには、一般的なプログラミング言語にあるオペレータやオペランドなどの命令のフィールドという概念はなく、命令は 0 から 255 までの値にそれぞれ割り振られている。したがってバイトコードによる中間コードは、大変コンパクトなものとなる。

ILBI で用いられるバイトコードの命令には、スタック操作命令、引数付き命令、ジャンプ命令の 3 種類がある。

3 ILBI の実装

ILBI はバイトコードを解釈、実行する、仮想機械であり、記憶部と解釈実行部により構成される。このうち記憶部は、プログラムカウンタやスタックポインタ等のレジスタと、プログラム領域や定数領域などの静的なデータ領域、スタック領域やヒープ領域などの動的なデータ領域とから成る。動的データ領域のうち、ヒープ領域がごみ集めにより再利用される。ごみ集めにはコピー GC 法を用いた。

ILBI の実装には C++言語を用い、ISLisp のクラス構成をほぼそのままの形で用いて、ILBI のクラスを構成した。ただし、整数とキャラクタについてはクラスを作らず、Immediate data として扱った。また、他のオブジェクトと Immediate data を区別するために、ポインタタグという機構を採り入れた。

4. インタプリタの効率化

インタプリタは実行速度の遅さが最大の欠点である。そこで、命令の縫い糸コード化とスタックキャッシングを行い、実行速度の向上をはかった。

インタプリタは命令を実行する時、いったんループの先頭にもどってから、その命令コードのプログラム部分に分岐しなければならない。ここで、ループの先頭にはもどらず、直接次の命令に分岐をする効率化が、**縫い糸コード** (threaded code) 化である。

ループによるプログラムからの具体的なアルゴリズムの変更は、次の命令の実行に移る際に、いったん while ループの先頭に戻り switch 文により分岐を行っていたものを、goto 文により直接次の命令にジャンプするようにした点である。

またインタプリタでは、命令の引数アクセスにかかる時間も、インタプリタの性能を決定する一つの要因である。つまり、スタックと他のメモリ領域あるいはレジスタとの間での引数の受渡しが、インタ

ISLisp implementation

Junichi FUNATO, Atsushi MAEDA, Masakazu NAKANISHI

Department of Mathmatics, Faculty of Science and Technology, Keio University, 3-14-1 Hiyoshi, Yokohama, Kanagawa Pref., 223, Japan

表 1: 実行時間の比較 (秒)

	効率化なし		縫い糸コード化		キャッシング		両方の効率化を導入	
(ack 3 8)	24.93	(1.00)	20.25	(0.81)	22.11	(0.89)	18.58	(0.75)
(fib 30)	32.70	(1.00)	27.85	(0.85)	31.64	(0.97)	27.85	(0.85)
(tak 27 18 9)	108.80	(1.00)	88.15	(0.81)	101.39	(0.93)	87.16	(0.80)
bit	2.70	(1.00)	2.36	(0.87)	2.25	(0.83)	2.09	(0.77)

ブリタのオーバーヘッドとなる。そこで、スタック内の値をCPUのレジスタにキャッシングすることにより、引数アクセスにかかる時間を減らすことができる [2]。スタックの先頭の値を保持するレジスタをtosとする。例えば、1引数をとる命令を実行する場合を考えると、普通にスタックを用いる場合は、スタックから値を取り出し、演算し、結果をスタックに積むという作業を行う。一方スタックの先頭の値をtosに保持しておくと、スタックに対する値の積み降ろしが必要なくなる。スタックキャッシングを行うと、1オペランド命令や、関数呼び出しから戻るRETURN命令の実行速度が向上する。一方、スタックへの値の積み降ろしは遅くなる。

5. 実験結果

効率化を行わなかった場合、縫い糸コード化を行った場合、スタックキャッシングを行った場合、その両方を行った場合とで、実行時間の比較を行った (表 1) [1]。

命令の縫い糸コード化では、いずれの場合も実行時間が短縮されている。この結果から、縫い糸コード化によりインタプリタの効率が向上することが確かめられた。

また、tosを導入することにより、いずれも実行時間が短縮されている。しかし、短縮の度合は実行するプログラムによって差がある。これはtosの導入により実行が速くなる命令と、逆に実行が遅くなる命令が存在するためである。したがって、RETURN命令を頻繁に行う再帰のプログラムや、1オペランド命令を多用するプログラムは、tosの導入により実行速度の向上が期待できる。一方、プログラムの書き方によってはtosを導入した方が遅くなる場合があることが確認された [3]。

6. 結論・今後の展望

本研究では、ISLisp処理系のバイトコードインタプリタに、命令の縫い糸コード化と、スタックトップの値をレジスタに保持するスタックキャッシングによる効率化を採り入れ、その効果を検証した。

まず命令の縫い糸コード化を行うことで、実行速度が向上し、インタプリタの効率化がはかれることが確かめられた。また、スタックキャッシングでは実行するプログラムにより多少の差はあるが、実行速度が向上することが確かめられた。ただし、プログラムの記述の仕方によっては、スタックキャッシングを行うことにより、逆に実行速度が遅くなる場合があることが確かめられた。

今後の展望としては、まずISLispのオブジェクト指向機能に対応するための、バイトコードの設計、追加を行う。さらに、より効率化をはかるために、動的スタックキャッシングを採り入れ、実行速度の改善をはかる。

参考文献

[1] *Programming Language ISLISP Working Draft 11.4*, ISO/IEC JTC 1/SC 22/WG 16., July 1994.

[2] Etrl, M.A. *Stack Caching for Interpreters, ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, 315–327, 1995.

[3] 船戸 潤一. スタックマシンによるバイトコードインタプリタの作成学士論文, 慶應義塾大学, 1996.

[1] かっこの数値は効率化なしの場合との実行時間の比率である。