

5Y-3

Y A L L (Yet Another Lexical scope Lisp) について

吉田 信博 (東芝)、武林 圭吾 (東芝コンピュータエンジニアリング)

1. はじめに

Lexical scopeの規則に基づいたdeep-binding-techniqueを用いたLISPシステムを実現させた。本報告では、実行速度向上のための手法を中心として述べる。

2. upwardのfunargの必要性

upwardのfunargとは、環境を現在よりも上位フレームに保障する。

downwardのfunargとは、環境を現在よりも下位フレームに保障する。

YALLにおける静的クロージャはdownwardのfunargをサポートしているが、upwardのfunargはサポートしていない。

upwardのfunargをサポートするためには関数の評価毎にclosureを作成する必要がある。また、その環境を保存するには、upwardではa-listを、downwardではa-frameを用いる必要があり、メモリの使用量が増え、評価時の実行速度が遅くなる。

upward funargは関数内のローカル変数を静的変数として用いることが主眼と考えられる。この目的のためには、packageの概念を拡張し、関数に対して自動的にpackageを割付け、その関数のlocal static variableをpackageのstatic変数として扱う(MacLispのclosureと同様)ことにより、事実上何等問題は発生しない。

以上からも明らかにlocal static variable宣言(def un時に処理され、interpret時に無視される)を導入すると、事実上、upward funargの問題も殆ど解決され、実行速度も充分期待できる。

3. semanticsの差について

lexical scopeの採用でupward frame/lexical semanticsの差は完全になくすることが可能である。

binding法はshallow bindingではなく、stack bindingを採用し静的スコープを実現している。従ってupward frame/lexical semanticsの差は発生せず、また、stack bindingはshallow bindingに比べ、4-6%程度の性能低下でしかない。

4. データ表現

LISP Objectは多くの型を持ち、この型の判定は実行時になされなければならない。実現法としては、LISP object内に型指示子を含ませる方法、また、LISP Objectを指すポインタに型情報を持たせる方法の2つが考えられる。前者はメモリのアドレス割付けに関して自由度が高いが、型の指示子を認識するまでに後者に比べ最小でも1メモリ以上必要とする。後者の欠点は、前者の長所

であるアドレスの自由度が下がることにあるが、充分大きな仮想アドレスを持つ場合、ある程度型によってアドレス域を固定化したとしても、事実上問題は発生しない。

また、garbage collectionの効率を考慮した場合、同一の性質を持つObjectが一定の場所に集中していることが望ましい。

従って、アドレス表現そのものに型情報を持たせるべきと考える。

4.1. アトム

シフト長の印字名及び文字列は24ビットのhash keyで一意に決定可能である。

整数は上述の充分に大きな仮想アドレスの領域を用いている。整数の範囲として(- (expt 2 29))から(expt 2 30)まで使用可能である。これによりすべての整数に対して値を比較する場合、関数equalではなく、関数eqで比較可能である。

無限多倍長は現時点では実装していない。

4.2. consセル

1 cons LISPアドレスはcar部32bits、cdr部32bitsの合計64bitsの構造体となっており、car部の下位3bitsはgarbage collection用のマークビット、その他の目的で使用する。

5. 実行時システム

5.1. インタプリタ

システムは移植性を考慮すると高級言語で記述することが望ましい。しかし実行速度の点ではupward frameを直接操作でき、また、upward frame単位を越えて、upward frameの共通割付けが可能でアドレス比較すると難点がある。

ここでは、実行速度の重視の立場からupward frameを選択するが、移植性を考慮し、upward frameと比較的小数のupward frameを持った仮想マシンを定義しこのupward frame(LAP)を用いる。

仮想マシンと実マシンとの対応(LAPからnative assemblerへの変換)は、upward frame展開によってなされる。(PSLと同様の思想)

5.2. コンパイラ

upward frameも前述のLAPをupward frameとして出力する。cadr、基本述語、固定長整数演算、map関数族はupward frame展開され、lsubr関数族は2引数を持つ基本内部関数の組合せに展開される。

また、upward frame割付け/多重upward frame除去等のpeep hole optimizeは、LAP生成時とLAPからnative assembler

への展開時の2段階でおこなわれる。

5.3. ローダ

操作性の点からインタプリタを実現した。native assemblerからロートマシン形式までへの変換は、対象システムにあるassembler/linkerをそのまま使い、参照関係の解決はインタプリタでLINKER出力のmapを参照しながら行う。(システム依存となる)

6. 実行速度の評価

システムの評価として(bita '(1 2 3 4 5 6 7 8))(tak 15 10 5)、Derivative、Data_Driven Derivativeを用い、インタプリタとコンパイラ双方についての実行時間測定結果を以下に示す。

表 1 インタプリタ/コンパイラの実行速度比

		インタプリタ使用	コンパイラ使用	インタプリタ/コンパイラ
bita 8	YALL	1150 ms	280 ms	4.11
	KCL	8590 ms	1630 ms	5.27
	YALL/KCL	0.13	0.17	0.77
tak	YALL	1980 ms	220 ms	9.00
	KCL	15084 ms	420 ms	35.91
	YALL/KCL	0.13	0.50	0.25
deriv	YALL	24390 ms	6320 ms	3.86
	KCL	123450 ms	13190 ms	9.51
	YALL/KCL	0.19	0.48	0.41
dderiv	YALL	26150 ms	7690 ms	3.40
	KCL	145260 ms	14710 ms	9.87
	YALL/KCL	0.18	0.52	0.34
Total	YALL	53670 ms	14510 ms	3.70
	KCL	294384 ms	29950 ms	9.83
	YALL/KCL	0.18	0.48	0.38

YALL/KCLとも、i80386CPU、16MHz(J3100SGT)

7. 考察

表1からも明かに、インタプリタとコンパイラの実行速度比は、KCLの方が大きい。

この差はupwardのfunargをサポートしているため、KCLのインタプリタの実行速度が遅くなったことによると考えられる。

7.1. KCLにおけるclosure

コンパイラの場合、解析時closureを判定し、必要な環境のみを保存するが、インタプリタの場合、実行しないと判定できず、常に環境を保存せねばならぬため、上記の差がでてくると考えられる。

このため、インタプリタとコンパイラの実行速度比は大きくなる。

7.2. YALLにおけるclosure

前述したように、YALLではupwardのfunarg機能を削除したため、インタプリタにおいて評価する毎にclosureを作成する必要はない。

従って、インタプリタとコンパイラの実行速度比は小さく、インタプリタの性能が上がっていると考えられる。

7.3. 各評価プログラム毎のインタプリタ/コンパイラの実行速度比

7.3.1. 評価関数bitaについて

YALLとKCLの実行速度比は、ほぼ同じである。

これは、downwardのfunargしか使っていないためであると考えられる。

7.3.2. 評価関数takについて

YALLとKCLでは、実行速度比が4倍程度の差がある。

関数takはfunction callの回数が多い。

そのためKCLのインタプリタにおいて、常に環境を保存せねばならず、インタプリタの速度が遅くなり、この差が顕著に出ると考えられる。

(tak 15 10 5)において関数tak、1-をcallする回数を表2に示す。

表 2 (tak 15 10 5)におけるfunction call count

関数名	回数
tak	10345
1-	7758
Total	18103

8. おわりに

今後の課題として、以下の点がある。

8.1.expr、fexpr等のLISPで記述された関数からコンパイルされた関数をcallできる。しかし、コンパイルされた関数からLISPで記述された関数をcallできない。これをcall可能にする。

8.2.multiple value、無限多倍長のサポート

8.3.step等のdebug機能の充実

8.4.本LISPシステムの他のシステムへの移植を行い、portabilityを評価し、マクロ(LAP)の改善を計る。

9. 参考文献

- [1] Richard P. Gabriel: Performance and Evaluation of Lisp Systems, MIT Press
- [2] Guy L. Steele Jr.: Common Lisp the language, Digital Press
- [3] Taiichi Yuasa and Masami Hagiya: Kyoto Common Lisp Report, TEIKOKU INSATSU INC.