記　号　処　理 53－3
（１９８９．１１．２２）

UtiLisp/C のインタプリタ

田中 哲朗
東京大学工学部

UtiLisp/C は、メインフレームや MC68000 系の機械で広く使われている UtiLisp を C 言語で書き直した処理系である。UtiLisp の言語仕様はインタプリタの実行速度を重視したものになっている。そのため、UtiLisp/C も実行効率を重視して設計された。UtiLisp/C は SPARC プロセッサ上での使用を想定しており、SPARC 上で走らせる時は、SPARC 固有の機能を生かして高速な型チェックを行なうようにしている。この結果、UtiLisp/C は高級言語で書いたにもかかわらず十分な実行速度を得ている。

The UtiLisp/C Interpreter

TANAKA Tetsurou
Faculty of Engineering, University of Tokyo
7-3-1, Hongo, Bunkyo-ku, Tokyo, 113 Japan

The UtiLisp is a dialect of Lisp, which has been implemented on main frame computers and MC68000 series processors. We made a UtiLisp system named UtiLisp/C in C language. The design of the UtiLisp/C is determined with a care of speed of interpreter. Since the UtiLisp/C takes advantage of special features of SPARC processor when checking types of Lisp objects, the executing speed on SPARC processor is fast enough for usual works.

1 はじめに

UtiLisp[1] は、はじめメインフレーム用の Lisp 処理系として開発されたが、後に MC68000 用 [2]、MC68020, VAX 用 [3] の処理系も作られた。これらの処理系は、実行効率を重視してアセンブラで記述されてきた。そのため、新しいプロセッサがでるたびに処理系を作り直さなければならなかった。

そこで、今回 SPARC プロセッサ用の UtiLisp インタプリタを作る際は、使用言語として C 言語を選んだ。しかし、UtiLisp の伝統であるインタプリタの実行効率も無視する訳にはいかない。そこで、SPARC 上で走らせることを考慮して型表現等を決定した。

以下では、UtiLisp/C の内部構造について詳しく説明する。

2 型の表現

UtiLisp/C の型表現は 表 1のようになっている。

表 1: UtiLisp/C の型表現

型	ポインタタグ	オブジェクトタグ (32bit 中下位 6bit)
固定長整数	0000	
シンボル	01	
コンス	10	
浮動小数点実数	11	000100
可変長整数	11	001100
配列	11	011100
文字列	11	100100
ストリーム	11	101100
コード	11	110100

UtiLisp/C ではヒープ領域をワード単位で確保するのでリスプオブジェクトの先頭アドレスは下位 2 ビットが必ず 00 になる。この部分をタグとすることによって、固定長整数、シンボル、コンスを区別することができる。他のリスプオブジェクトは先頭の 1 ワードのオブジェクトタグによって区別する。オブジェクトタグによる型チェックは速度が劣るが、これらの型はそれほど頻繁に参照されないので、全体の速度にはそれほど影響がない。

SPARC では。ワード境界をまたがったデータを一度にアクセスしようとするとトラップが生ずる。UtiLisp/C では、この性質を使ってリストやシンボルを扱う組み込み関数において、型チェックを省略している。

例えばコンスはヒープ上に car 部と cdr 部の 2 ワードの実体を持っているが、コンスを指すポインタは、タグの部分の 10 つまり 2 バイトが足されたアドレスを指している。car は引数としてコンスだけを許しそれ以外はエラーとするという関数であるが、本来なら引数の下位 2 ビットを取ってそれが 10 以外の時はエラー処理関数を呼び出し、10 の時は引数から 2 を引いたアドレスの内容を返すように書かなければいけない。しかし、SPARC 上で走らせる場合は、下位ビットのチェックを省略していきなり、引数から 2 を引いたアドレスを参照しても構わない。引数がコンスでないときはそこでバスエラーのトラップが生じるからである。

ただ、固定長整数のようにヒープ上に実体をもたない Lisp オブジェクトはこの方法を使って型チェックを省略することができない。しかし、SPARC にはこれを考慮したタグつき演算命令が用意されている。これは、ソースデータのどちらかの下位 2 ビットが 00 でないときはトラッ

プが生ずるという加減算命令である。UtiLisp/Cでは、これを使って固定長整数を扱う組み込み関数の型チェックを行なっている。

3 メモリ管理

UtiLisp/Cの扱うLispオブジェクトは、図1のようにヒープ領域と4本のスタックに置かれる。

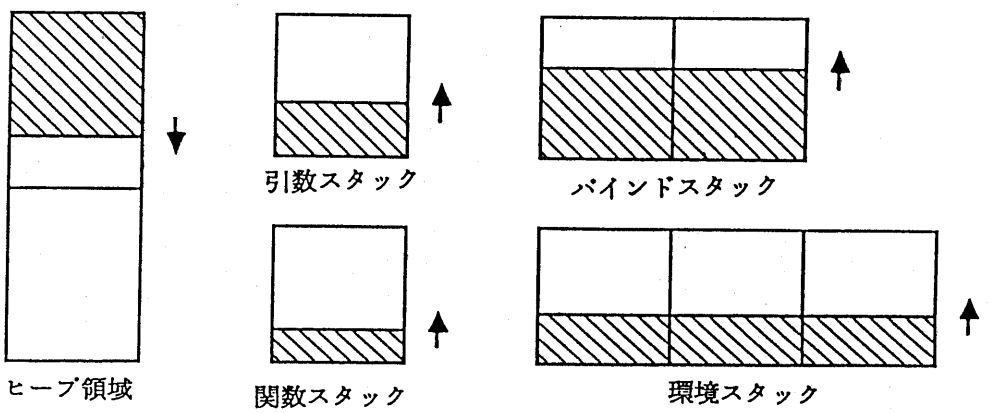

図 1: UtiLisp/C のメモリマップ

ヒープ領域は、コピー方式のGCを採用しているので、同じ大きさの2つの領域に分かれている。普通に実行している時は片方の領域しか使われない。使っている領域に空きがなくなったら、もう片方の領域にゴミでないLispオブジェクトをコピーして、以降はそちらの領域を使う。GC中はアドレスによって、どちらの領域を指しているかを決定できるで、GC用のマークビットは必要ない。

ヒープ領域には、いろいろな型のリスプオブジェクトが混在している。いろいな大きさのリスプオブジェクトが入っているが、領域の確保はワード単位に行なわれる。

4本のスタックのうち、引数スタックは関数への引数を積むためのものである。このスタックは頻繁に使われるのでスタックポインタはアクセスの遅いグローバル変数ではなくて、Cの関数の引数として渡すようにした。

バインドスタックは、λバインドをする時にシンボルとシンボルの値を組にして積んで置くためのものである。UtiLispでは、浅い束縛を採用しているので、このスタックが必要になる。

関数スタックは、エラーが生じた時にどのレベルでエラーが生じたかを表示したり、backtraceという関数で関数の呼び出し関係を参照するために使われる。

環境スタックは、catchとthrow、loopとexit、progとgoなどの大域脱出をサポートするために使われる。

これまでのUtiLispでは、スタックは1本にまとめられていたが、そのために情報を区別するための余計なタグが必要になった。また、不用意にスタックに値を積むと、GC時に混乱を招くことがあった。C言語で記述する際にはこれらの反省から、用途によってスタックをわけていった結果、この4本になった。

この他に、nil や t など組み込み関数から参照されるシンボルはルートオブジェクトとして別のところに置かれる。

4 GC

UtiLisp/C ではコピー方式の GC を採用している。コピー方式の GC を用いるとヒープ領域の使用量が倍になるが、GC のアルゴリズムは単純になり、結果として GC にかかる時間も少なくなる。

コピー方式の GC を自然に記述すると再帰的に書ける。再帰的にコピーしていくと、参照が局所化するという利点があるが、最悪の場合はヒープ領域の大きさに比例した回数の再帰呼び出しが生じてしまう。これほど極端な場合でなくても SPARC の場合再帰呼び出しのオーバーヘッドが大きく、GC の速度の低下につながってしまう。

そこで、UtiLisp/C では再帰を使わないコピー方式の GC のアルゴリズムを採用した。これは、古いヒープから新しいヒープに Lisp オブジェクトをコピーする際に、その Lisp オブジェクトからの参照をその場で更新しないで、後で新しい領域を走査した時に初めて更新するという方法である。

この方法の場合、新しいヒープを走査する際に先頭ワードを見てそのオブジェクトの型が決定できなければいけない。オブジェクトタグが付いていないコンスやシンボルは、ここに古いオブジェクトへのポインタを書いておいてこの時点でコピーするようにした。

5 組み込み関数

組み込み関数の中には、引数の数がいろいろ変えられるものがある。そのためこれまでの UtiLisp の処理系では、組み込み関数のエントリーを複数作って、引数の数に応じて別のアドレスから関数の実行を始めるということを行なっていた。

しかし、C 言語の関数のエントリーは 1 つしかないので、この方法を使うと引数の数ごとに似たような別の関数を作らなければいけなくなる。これは、無駄なので、C の関数の引数として Lisp の引数の数を与え、関数の中で自分で引数の数をチェックしてエラー呼び出しを行なうようにした。

組み込み関数の記述の例として関数 car の定義をあげる。

```
WORD car_f(na,fp)
WORD *fp;
{
  WORD a;

  if(na!=1)parerr();
  return(car(checkcons(a,ag(0))));
}
```

引数の na は Lisp の引数の数で、fp というのは引数スタックのスタックポインタである。parerr, car, checkcons, ag というのはいずれも C のマクロであり、プロセッサの違いをここで吸収することができる。

SPARC の上では固定長整数を扱う関数の中でタグつき演算命令を使うが、これは C 言語で書くことができない。そこで、インライン関数を使うことにした。Sun の C コンパイラの場合は、

```
.inline _tadd,8
taddcctv %o0,%o1,%o0
.end
```

のように定義したファイルを作っておいて、コンパイル時に指定すると、tadd という名前の関数があるかのように扱うことができ、この部分が出力されるアセンブラファイル中に最適化された形で表れるというものである。

GCC にも同様な機能があるのでそれを使えばよい。このような機能がないコンパイラでも、アセンブラファイル中の call 命令を sed で置換することによって同様の効果を実現することができる。

6 エラー処理

通常のエラー呼び出しは関数呼び出しによって行なうが、SPARC 上で走らせる場合は signal によってエラー処理を行なう場合がある。この場合エラーの検出は楽だが、それ以降の処理でいろいろ工夫する点がある。以下では、それらの場合についての処理について説明する。

6.1 型エラー (1)

固定長整数以外の型エラーはすべてワード境界をまたがったワードアクセスによるバスエラーによって検出する。バスエラーが生ずると、SIGBUS の signal が発生する。signal ハンドラには、signal 発生時の種々の情報が渡されるが、そこから UtiLisp のエラー処理関数にエラーがどこで生じたかと、エラーの原因のオブジェクトを選び出して引数として渡す必要がある。

例えば、a というシンボルに対して car を実行しようとして生じた型エラーでは err:argument-type を a と C#car を引数として funcall しなければならない。

エラーがどこで生じたかは、関数スタックを見ればわかるので、後はエラーの原因となった Lisp オブジェクトを決定することができればよい。

SUN4 では、ワード境界を原因とするバスエラーが生じた時、signal ハンドラに、どのアドレスを原因としてエラーが生じたかの情報が渡ってこない。UtiLisp/C では SPARC が RISC であることを利用して、そのアドレスを決定している。

まず、signal 発生時の PC の内容を見れば、どの命令を実行しようとして、エラーが発生したかが分かる。バスエラーを生ずる以上、その命令はロード命令かストア命令である。幸いなことに SPARC は RISC なので、アドレッシングモードは、レジスタ＋オフセットとレジスタ＋レジスタしかない (図 2)。

Lisp オブジェクトの内容を参照する場合に使われるのはレジスタ＋オフセットのアドレッシングモードだけである。その場合シグナル発生時のそのレジスタには、Lisp オブジェクトそのものが入っている。後はそのレジスタの保存された値を引数にしてエラー処理関数を呼び出せばよい。

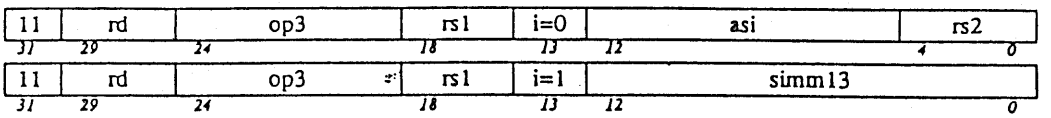

図 2: ロードストア命令のフォーマット

6.2 型エラー (2)

taddcctv, tsubcctv は、内容のどちらかの下位2ビットが00でないときにエラーを起こすが、両方とも00であって演算の結果、オーバーフローが生ずる時も同様にエラーを生ずる。

signalハンドラは、足そうとしたレジスタ (片方が即値のこともある) の内容を調べて、どちらかのタグが00でないときは、その内容をもってエラー処理関数を funcall する。両方が00でないとき、つまり複数の引数について型エラーが生じた時は、どちらのエラーを表示するかは、処理系製作の自由度として残されている。どちらのタグも00の時は、オーバーフローを起こした時である。この時は、UtiLisp ではエラーとされていないので、そのままリターンすればよい。

6.3 引数チェック

UtiLisp/C ではほとんどの組み込み関数の中から、エラー関数が呼ばれる可能性がある。大部分の組み込み関数は他の C の関数を呼び出さない、いわゆる末端関数である。Sun4 用の C コンパイラは末端関数については、レジスタウィンドウをずらさないで、手持ちのレジスタだけでやりくりする良いコードを出してくれるが、エラー関数を呼び出す可能性があるために末端関数ではなくなってしまうと、効率の低下につながる。

そこで、引数の数が違うというエラーも signal を使って処理するようにした。これは、単に引数の数が違う時は、違法なアドレスをアクセスしてそれによるセグメントバイオレーションのエラーによって捕まえるというものである。セグメンテーションバイオレーションをおこすアドレスというのはマシンに依存するからこれはあまり一般的な方法ではない。SPARC 以外の機械では末端関数にすることのメリットがないので関数呼び出しによってエラー処理を行なう。

7 評価

インタプリタの速度だが、これは当初の目的が UtiLisp32 よりは遅くならないものにしようというものだったので、UtiLisp32 と比較することにした。この結果が表2である。ベンチマークに使用した問題は、Lisp コンテストで使われた問題で Lisp のベンチマークでは良く使われる問題である。

使用した機種が SUN3/50 と Sparc Station だから、比較にならないのは明らかだが、どの問題でも UtiLisp32 以上なので、システムとして実用になる性能だといえるだろう。

8 おわりに

高級言語で処理系を記述する場合、一般的には実行するプロセッサを意識しない。そのため、アセンブラで書いた場合と比較して、プロセッサの性能を出し切ることができず、実行効率が落ちるのが普通である。

UtiLisp/C では、C 言語での記述で移植性を考慮しているものの、SPARC プロセッサ上で使う時は、SPARC の特徴を生かすように作られている。その結果、十分な性能を得ることができた。

この処理系は今のところ、いくつかの場所でテスト的に使用しているが、処理系が安定したら広範囲に配布を開始する予定である。

参考文献

[1] 近山 隆： Lisp 処理系の構成法に関する研究, 1981 年度東京大学工学部情報工学博士論文 (1982).

表 2: ベンチマーク表

番号	プログラム名	UtiLisp32 (Sun3/50) 1/60 sec	UtiLisp/C (Sparc Station 1) 1/60 sec	速度比
1-1	tarai	1183	417	0.35
1-4	tak	658	233	0.35
2-1	list-tarai	205	93	0.45
2-2	srev	651	298	0.46
2-4	qsort	685	317	0.46
2-5	nrev	501	303	0.60
2-6	reverse	83(184)	48(42)	0.58
2-7	nreverse	42(182)	22(42)	0.52
3-1	string-tarai	2296(462)	898(139)	0.39
4-1	flo-tarai	1431(143)	472(34)	0.33
4-2	big-tarai	1664(84)	710(18)	0.43
5-1	bubblesort	284	152	0.54
6-1	sequence	52	19	0.37
7-1	(bita)	141	93	0.66
7-3	(bitb)	517(74)	348(17)	0.67
9-1	(tpu)	66	34	0.52
10-1	(prolog)	128	66	0.51
11-1	(diff)	149	57	0.38
12	(test)	7823(648)	3890(218)	0.50

[2] 和田 英一, 富岡 豊 : UtiLisp の MC68000 への移植, 情報処理学会記号処理研究会資料 SYM29-3(1984).

[3] Kaneko, K. and Yuasa, K.: A New Implementation Technique for the Utilisp System, Preprints of WGSYM Meeting, IPS Japan, 87-41 (1987).

[4] Steven S. Muchnick : SPARC の最適化コンパイラ (日本語訳), ユニックス・マガジン vol.4, No. 1, pp. 87-102(1989).

[5] 田中 哲朗 : RISC 向きの Lisp 処理系の製作, 1988 年度東京大学工学部情報工学修士論文 (1989)

[6] 村松 正和 : UtiLisp/C のコンパイラ, 1988 年度東京大学工学部計数工学卒業論文 (1989).