Vol. 32 No. 5 情報処理学会論文誌 May 1991

SPARCの特徴を生かしたUtiLisp/Cの実現法

田 中 哲 朗†

この論文では，SPARCプロセッサの上で実行するために開発したUtiLisp/Cの実現法について，型チェックとエラー処理を中心に述べる．Lispの実行には動的な型チェックが必要である．型チェックはLispの実行時間のかなりの部分を占めるので，高速なLisp処理系を作るには効率的な型チェックを実現しなければならない．実行速度を重視して言語が設計されているUtiLispは，特にこの点を考慮して処理系が作られてきた．今回，SPARCプロセッサ上の処理系を作るにあたっても，そのことに留意してSPARCプロセッサ固有の機能を使った高速な型チェック機構を実現した．そのため，高級言語で書いたにもかかわらず充分な実行速度を得ることができた．

Implementation of UtiLisp/Cwith SPARC Processor

TANAKA Tetsurou†

1. はじめに

UtiLisp[1] はLisp言語の一方言であり，実行速度を重視して設計されている．

最初の処理系はメインフレームのアセンブリ言語で記述された．他のプロセッサへの移植は難しいと思われたが，最初に汎用レジスタの数やアドレス空間の広さなどメインフレームと多くの共通点を持つMC68000への移植(UtiLisp68)が実現した[2]．

その後，これらのプロセッサと違って32ビットすべてをアドレス指定に用いるプロセッサが作られるようになった．このようなプロセッサではそれまでのUtiLispで用いていた型チェックやヒープ管理法を用いることができない．そこで，言語仕様を一部拡張し根本から作り直したUtiLisp32という処理系が作られた[3]．

UtiLisp32はアセンブリ言語で書かれていたものの，LAP形式によるマクロをうまく利用してプロセッサ間の違いを吸収し，MC68010/20，VAXなどのいくつかのプロセッサ上で実現された．

近年になってRISCアーキテクチャに基づくプロセッサが登場し，32ビットプロセッサの主流になった．これらのプロセッサは高級言語で書かれたプログラムを高速に実行するように命令セットが設計されている．そのため，高級言語でプログラムを書いても，人間がアセンブリ言語で巧妙に書いたプログラムとの実行時間の差はそれほど大きくない．

そこで，RISCプロセッサの一つであるSPARCプロセッサ用のUtiLisp処理系を作るにあたって，記述言語はこれまでの処理系のようにアセンブリ言語ではなく，デバッグの容易さや移植性を重視してC言語とすることにに決めた．また，UtiLisp32の型チェック法はC言語では効率的に実現することができないため，C言語で容易に書け，SPARCプロセッサの特徴を生かした別の型チェック法を用いることにした．

このような方針によって作られた処理系がUtiLisp/Cである．UtiLisp/Cに関してはSPARCのアセンブリ言語の特徴を生かしたbignumルーチン[5]や，各種のオプティマイズを行なっているコンパイラ[7]などさまざまな話題があるが，この論文ではUtiLisp/Cインタプリタの主要部分に話題をしぼって述べる．

2. 型の表現

2.1 旧UtiLispにおける型表現

メインフレーム版のUtiLisp及びマイクロプロセッサ用のUtiLisp68では，アドレス空間が24ビット分しかないことを利用して，32ビットデータの上位8ビットをオブジェクトの型を表すタグとして使った．その際タグの割当を巧妙に行なったため，型チェックを高速に行なうことができた．

UtiLisp32では，アドレス空間が32ビットに広がり，この方法を適用することができなくなった．アドレス空

† 東京大学工学部
Faculty of Engineering, University of Tokyo

間の下位 24 ビットだけを使うことを前提にしてこの方法を使うことも可能だが, その場合メモリの参照のたびにタグ部分をマスクする命令を書かなくてはいけない. これは効率に大きく影響する. そこで, 新たな型表現を考え出すことになった.

色々検討した結果, UtiLisp32 ではヒープ分割法を使うことになった. これは, シンボル, コンス, それ以外のヒープ上に実体を持つオブジェクト (others) をヒープ上の別々の領域に置き (図 1), アドレスの大小の比較によって型を決定する方法である. シンボルとコンスはこの方法だけで型を決定することができる.

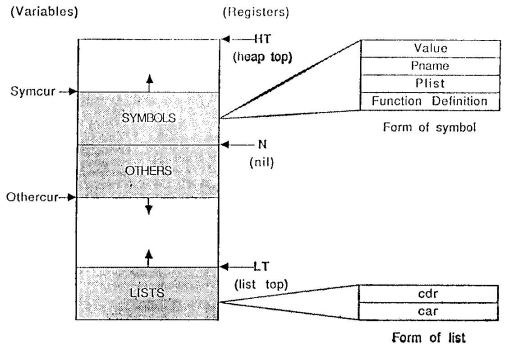


図1 UtiLisp32 のヒープ分割
Fig. 1 Heap allocation of UtiLisp32

ヒープ上に実体を持たない固定長整数は, ヒープとは絶対に重ならないよう上位 2 ビットにタグを付け, さらに下位 2 ビットを除いた 28 ビットに実体を入れる. others にはさらに型を区別するためにオブジェクトの先頭に 1 ワードのオブジェクトタグをつける.

UtiLisp/C でも UtiLisp32 の型表現法を継承することは考えられたが, C 言語では領域の境界のアドレスをレジスタに入れておくことができず*, 型チェックのたびにメモリ参照が生じてしまい, 実行速度に大きく影響する. そのため, 別の方法を取ることにした.

2.2 下位ビットをタグとする方法

UtiLisp ではヒープをワード単位で確保するから, オブジェクトの先頭アドレスの下位 2 ビットは必ず 00 になる. ポインタの下位 2 ビットは無意味であるから, この部分をタグとして扱えば 4 種類の型を表現できる.

この方法は, ポインタの上位にタグをつける場合と違い, タグ部分を容易に取り出すことができ, メモリを参照する時はマスクを取るかわりに小さなオフセットをつけるだけでよいという点で優れている. C 言語で記述する際もマクロを使えばきれいに書くことができる. これらの利点を重視し UtiLisp/C ではこの方法を採用することにした.

この方法は UtiLisp32 の設計の際にも考慮されたが, 採用に至らなかった. その方法を今回採用したのは, RISC プロセッサ上で動かすことを想定しているためである.

CISC プロセッサではメモリ参照の際にオフセットが付いていると, オフセットがない場合と比べて命令の実行時間が長くなる. また, オフセットつきの場合は使えないアドレッシングモードもあるため (オートインクリメントなど), オフセットなしの場合は, 一命令ですむ時に, 複数の命令が必要になることがある. これではマスクを取る代わりにオフセットをつけて参照できるという利点を生かすことができない.

その点, RISC プロセッサはアドレッシングモードが少なく, オフセットが付いている場合とない場合との差がないものが多い. そのため, 下位ビットをタグとして使っても, タグがついていない場合との参照時間の差は生じない.

さらに, SPARC や一部の RISC プロセッサのようにワード境界をまたがったメモリ参照を許さないプロセッサでは都合がよいことがある.

通常, 特定の型の引数しか想定していない組込み関数は, 実行時に引数が然るべき型かをチェックし, 結果に応じてエラー処理ルーチンに分岐するようなコードを入れておく必要がある.

しかし, SPARC のようなプロセッサの場合は, 型チェックをした後でそのオブジェクトの内容を参照するならば型チェックのためのコードを入れる必要がない. 想定していない型の引数の場合はタグを打ち消すオフセットを付けたアドレスが, ワード境界とずれてしまい, メモリ参照でトラップが生じ, ここで型エラーが検出されるからである.

2.3 UtiLisp/C の型表現

決定した UtiLisp/C の型表現を表 1に示す.

下位 2 ビットだけで区別することのできるのは 4 種類のオブジェクトだけである. UtiLisp の型すべてを表現することができないため, 使用頻度の小さいオブジェクトではオブジェクトタグも併用する.

UtiLisp32 にならって, ポインタタグだけで判別することのできる型は, 固定長整数, シンボル, コンスの三つとした.

固定長整数はメモリ上に実体がないためワード境界トラップを利用した型チェック法を使うことができない

*GNU C コンパイラを使えばグローバルなレジスタ変数を使うことができる

表1 UtiLisp/C の型表現
Table 1 Internal representaion of Lisp objects

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

が, SPARC にタグつき加減算命令というのがあるのでそれを利用した型チェックを行なう. SPARC のタグつき加減算命令とは, オペランドのどちらかの下位 2 ビットが 00 でない時はトラップを生ずるという命令である. この命令のため固定長整数のタグは自動的に 00 に決まる.

UtiLisp/C ではさらに 2 ビット分をタグとしている. これはオブジェクトタグと区別するために 1 ビット必要であり, ついでに 1 ビット削って UtiLisp32 と互換にしたためである.

3. メモリ管理

UtiLisp32 はヒープ分割法を用いている. ヒープは三つに分かれており, シンボル, コンス, others がそれぞれ別々の領域を占めている. それに対し, UtiLisp/C では, 前章で述べたような型表現を用いているため, ヒープは一つにまとめられて, いろいろな型のオブジェクトが混在している.

UtiLisp の処理系では, 高速化のためヒープ以外にスタック上にもオブジェクトを置くようにしている. これまで作られた UtiLisp の処理系はいずれもハードウェアのスタックをそのまま利用して 1 本のスタックに, サブルーチンの戻り番地や Lisp の引数, 束縛情報などを混在して置いてあった.

UtiLisp/C は C で記述したため, スタックの中身を全部管理することができなくなった. C で書かれた多くの Lisp 処理系は, Lisp オブジェクトをハードウェアスタックに入れておくために, スタック上にリンク構造を作るようにしている. これでは, スタックを使うことの利点が失われてしまうので, Lisp オブジェクトだけを収めるスタックをソフトで用意することにした.

スタックにはさまざまなものを積む必要があるが, UtiLisp/C ではスタックに積むオブジェクトの性格を 4 通りに分け, 引数, バインド, 関数, 環境と四つのスタックを用意することにした (図 2).

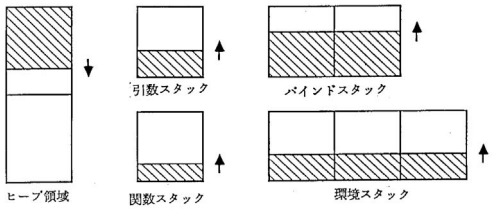


図2 UtiLisp/C のメモリマップ
Fig. 2 Memory map of Utilisp/C

引数スタックは関数への引数を積むためのスタックで, 他に eval や組み込み関数の中で, GC の際に保存しておくローカル変数を保持するためにも使われる. 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.

バインドスタックには, λ バインドをする際にシンボルとシンボルの値を組にして積んで置く. このスタックのスタックポインタは, アクセスの遅いグローバル変数だが, 続けて参照するときには, 一時的にローカルなレジスタ変数にコピーして使うことによって, 高速化を図っている.

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のに使う. コンパイルされたコードの場合は, 関数スタックの先頭が実行中の関数であることを利用し, このスタックを介してリテラルの参照を行なう.

環境スタックは, `break` と `unbreak`, `catch` と `throw`, `loop` と `exit`, `prog` と `go`, `return` などの大域脱出をサポートするために使われる. UtiLisp/C では UtiLisp32 のように大域脱出の際に, スタックの内容をしらべてコンテクストの回復を行なうことができない. あらかじめ setjmp を行なっておいて脱出時に longjmp を行なう. 環境スタックの三つ組の一つには, jmp_buf へのポインタを入れておく.

この他に, `nil` や `t` など組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く.

4. ゴミ集め (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いので，GCの速度低下を引き起こしかねない．

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．

ヒープの残りが少なくなってくると，頻繁にGCが起き，実行速度が低下する．そのため，UtiLisp/CではGC後のフリー領域がヒープ全体に占める割合が，ある程度小さくなったら，ヒープを倍の大きさに確保し直して実行を再開するという機能をつけた．

5. 組込み関数

組込み関数の中には，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，組込み関数のエントリーを複数作り，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ようにしていた．

しかし，C言語の関数のエントリーは一つしかなく，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．これは実行ファイルのサイズを大きくしデバッグを困難にする．そこでUtiLisp/Cでは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出しを行なうようにした．

組込み関数の記述の例として関数`car`の定義を示す．

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

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

引数naはLispの引数の数で，fpは引数スタックのスタックポインタである．parerr, car, checkcons, ag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．

SPARCでは固定長整数を扱う関数の中でタグつき加減算命令を使うが，これはC言語で書くことができず，インライン関数を使うことにした．Sun標準の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き，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る．この部分が出力されるアセンブリ言語ファイル中に最適化された形で表れる．

GNU Cコンパイラでは，インライン関数中にasm文で記述することにより，同様のことを実現することができる．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で置換すれば最適化はなされないものの同様の効果を実現できる．

6. エラー処理

通常のエラー呼出しは関数呼出しによって行なうが，signalによってエラー処理を行なう場合もある．この場合エラーを検出した後，エラー処理を行なう際のいろいろな工夫が必要になる．以下にそれらの場合処理について述べる．

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，OSに依存する形で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の生じた場所と，エラーの原因のオブジェクトを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`a`と`C#car`(`car`のコード)を引数として`err:argument-type`を`funcall`しなければならない．

エラーの生じた場所は，関数スタックを見ればわかる．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，エラーを生じたアドレス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の内容から，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. 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 3).

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.

6.2 型エラー (2)

タグつき加減算命令 taddcctv, tsubcctv は, 両方のオペランドの下位 2 ビットがともに 00 でないときにエラーを起こすが, 両方とも 00 であっても演算の結果, オーバーフローが生ずると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, 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, 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. タグが両方とも 00 の時は, オーバーフローと分かる. UtiLisp では固定長整数のオーバーフローはエラーにしていないため, そのままリターンすればよい.

6.3 引数チェック

UtiLisp/C ではエラー関数はほとんどの組込み関数の中から呼ばれる可能性がある. 大部分の組込み関数は他の C の関数を呼び出さない末端関数である. 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高速な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. 引数の数が違う時は, 特定の違法なアドレスをアクセスしてトラップを生じさせてエラー処理ルーチンに飛ぶのである. 末端関数にすることのメリットがない場合は通常通り関数呼出しによってエラー処理を行なう.

7. 評 価

7.1 実行速度

実行速度の評価は同じプロセッサ上の他の処理系との比較で行なうのが一般的だが, 同じ Lisp 処理系とはいっても言語仕様の違いによってインタプリタの速度が大きく違ってくるので比較にならない. そもそも設計の基本方針は, UtiLisp32 よりは遅くならないということだったので, UtiLisp32 と比較することにする.

ベンチマークの結果を表 2に示す. ベンチマークに使用したのは, 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ている.

Sun3/50 と Sparc Station 1(SS1) というプロセッサの能力に大きな差のある計算機を使っただけに, これだけ見て UtiLisp/C の設計方針の正当性を主張するのは誤りである. しかし, 当初の方針通り, どの問題でも UtiLisp32 を超える性能を示しているから, 実用的な速度は達成しているということはいえるであろう.

表2 ベンチマーク表
Table 2 Benchmarks

番号	プログラム名	UtiLisp32 (Sun3/50) 1/60 sec	UtiLisp/C (SS1) 1/60 sec	時間比
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

7.2 移 植 性

7.2.1 移植の問題点

UtiLisp/C は SPARC 上で動かすことを目的として作られたが, 多倍長整数を扱うルーチンを除けば, ほとんど C 言語で書いてある. そのため, 他のプロセッサ上でも少々の問題を解決すれば, 多倍長整数を除いた仕様で動かすことができる.

移植の際に問題になるは次のような点である.

- タグつき演算命令
  今のところ, タグつき演算命令を用意しているのは SPARC だけなので, それ以外のプロセッサに移植するには, 代わりにタグをチェックしてから演算するマクロ又はインライン関数を使う. タグつき演算

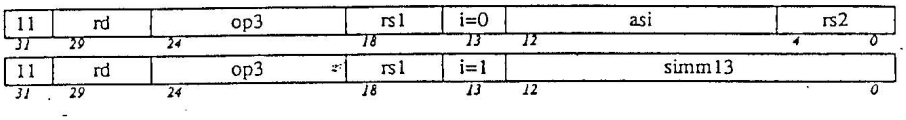


図3 ロードストア命令のフォーマット
Fig. 3 Format of load/store instructions

命令を持つプロセッサが出現したとときは, アセンブリ言語命令を C 言語から使う方法がコンパイラによって異なるため, それに対応しなければならない.

- ワード境界トラップ
  32 ビット境界がない場合は型チェックにワード境界トラップを使うことができない. その場合は明示的な型チェックを行なう必要がある. ワード境界トラップがあっても, シグナルハンドラの中から原因となった Lisp オブジェクトを特定することができなければ, 型チェックには使えない.
- インクリメンタルロード
  UtiLisp/C のコンパイラ[7] は, C 言語のコードを出力して外部の C コンパイラでコンパイルて, オブジェクトファイルを生成した後, それをメモリ上にロードするようになっている. つまり動的リンクの機構が必要になっているのである. 動的リンクは, BSD 系の unix では用意されているが, SystemV 系の unix には用意されていない場合もある. その場合は, ld の代替品を用意しなければならない. また, 動的リンクが用意されていても, オブジェクトファイルのフォーマットが異なる場合もある. この場合はロード部分のプログラムの変更が必要にある.

このように, make と入力するだけで移植ができるというわけではないが, 中身を知っていれば数時間で済むはずである.

実際に移植を試みたのは, MC68020, 80386, メインフレーム, R3000 などである. 移植の一例として R3000 への移植を次節で述べる.

7.2.2 R3000 への移植

RISC News に使われている R3000 は MIPS 社が設計した RISC プロセッサで, 多くの計算機に採用されている.

R3000 プロセッサは SPARC 同様ワード境界をまたがるアクセスを行なうとバスエラーを生ずる. これを利用して組込み関数での型検査を省略することができる. しかし, SPARC と違ってタグつき演算命令は用意されていない. そのため固定長整数演算の際には型チェックを必要とする.

R3000 用の C コンパイラはグローバルオプティマイズを行なう. 末端関数化すると最適化の役に立つので, 引数エラーを signal を使って検出するようにした.

移植の際には, a.out のフォーマットの違いからコンパイルされたコードのロード部分で書き直しが必要になったが, 他はほとんどヘッダ部分の変更に留まった.

表3 MIPS におけるベンチマーク表
Table 3 Benchmarks on R3000

番号	プログラム名	MIPS 1/60 sec	時間比 (対 SS1)
1-1	tarai	239	0.57
1-4	tak	136	0.58
2-1	list-tarai	46	0.49
2-2	srev	143	0.48
2-4	qsort	152	0.48
2-5	nrev	113	0.37
2-6	reverse	27(26)	0.56
2-7	nreverse	10(25)	0.45
3-1	string-tarai	483(60)	0.54
4-1	flo-tarai	263(17)	0.56
5-1	bubblesort	80	0.53
6-1	sequence	12	0.63
7-1	(bita)	45	0.48
7-3	(bitb)	146(8)	0.42
10-1	(prolog)	31	0.47
11-1	(diff)	34	0.60

実行速度は, 表 3のようにテストによってばらつきがあるもののおおむね Sparc Station 1 の 1.6 倍から 2.7 倍である. タグつき演算命令の使えない点で不利だと思われる tarai の場合でも, SPARC の 0.57 の時間で実行が終了する.

これは, Lisp のように再帰を多用する言語の実現にはレジスタウィンドウを使っている SPARC は向いていないということを表しているかも知れない.

8. 結　　論

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, アセンブリ言語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.

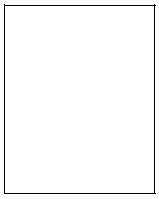
更に, 同じような特徴をそなえた RISC プロセッサ上へも容易に移植することができ, 高い性能を示すことが確認された.

謝辞 和田英一教授及び寺田実, 岩崎英哉, 湯浅敬, 小西弘一, 白川健治, 村松正和の各氏の協力に感謝する.

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).
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) 和田 英一 : UtiLisp/C の bignum ルーチン, 情報処理学会記号処理研究会資料 SYM53-4(1989).
6) 田中 哲朗 : UtiLisp/C のインタプリタ, 情報処理学会記号処理研究会資料 SYM53-3(1989).
7) 村松 正和 : UtiLisp/C のコンパイラの製作, 情報処理学会記号処理研究会資料 SYM53-5(1989).

(平成2年12月18日受付)
(平成3年 2 月12日採録)



田中 哲朗（正会員）

1965 年生まれ. 1987 年東京大学工学部計数工学科卒業. 1989 年同大学院修士課程修了. 現在同博士課程在学中. 記号処理言語, 漢字フォントに興味を持つ. ACM 会員.