

Tachyon Common Lisp の実現方式

五味 弘* 高橋 順一* 新谷 義弘** 伊藤 丹二** 長坂 篤**

*沖テクノシステムズ ラボラトリ

**沖電気工業 総合システム研究所

“COMMON LISP THE LANGUAGE SECOND EDITION” 準拠の高速かつ高い移植性を持つ Common Lisp 処理系である Tachyon Common Lisp の実現方式について述べる。本処理系は現在 i860TM RISC CPU を搭載した UNIX ワークステーション OKIstation 7300 上で動作する。Tachyon Common Lisp は実行速度を重視するため、i860TM CPU 特有の最適化を含む各種の高速化を行った。また、高い移植性を保証するためにその核言語のみを Lisp に似た構文を持つマクロアセンブラで記述し、その他の大部分を S 式で記述している。さらに大きなメモリ空間を使用でき、またそのカスタマイズが容易にできるメモリマネージャと、GC 時間の短縮のために圧縮領域を制御する GC を実現した。

Implementation Methods of Tachyon Common Lisp

Hiroshi Gomi* Junichi Takahashi* Yoshihiro Shintani** Tanji Ito** Atsushi Nagasaka**

*Oki TechnoSystems Laboratory, Inc.

8-10, Uchiyama 3-Chome, Chikusa-ku, Nagoya 464, Japan

**Oki Electric Industry Co., Ltd. Systems Laboratoris

11-22, Shibaura 4-Chome, Minato-ku, Tokyo 108, Japan

Tachyon Common Lisp is a full implementation of the Common Lisp described in the “COMMON LISP THE LANGUAGE SECOND EDITION”, and achieves high performance and high portability. Tachyon Common Lisp is currently implemented on OKIstation 7300 that is a UNIX workstation with an i860TM RISC CPU. We have taken many optimization methods including those for RISC CPU to realize high execution speed. The large part of Tachyon Common Lisp, except the kernel language, is written in S-expression for high portability. The memory manager can deal with huge memory space. It also allows users to configurate the memory space. Tachyon Common Lisp adopts a compactifying GC that determines optimum size of compaction subspace automatically for short GC run-time.

1 はじめに

Common Lisp の言語仕様の第2版として、“COMMON LISP THE LANGUAGE SECOND EDITION” (以降、CLtL2 と呼ぶ)[1]が出版され、いくつかの処理系が作成されている。CLtL2 は

- 1 “COMMON LISP THE LANGUAGE” 言語仕様の明確化及び変更が行われた
- 2 オブジェクトシステム CLOS が採用された
- 3 コンディションシステムが導入された
- 4 整形出力のカスタマイズが可能になった
- 5 LOOP マクロが大幅に機能向上されたなどの特徴がある。

本稿では CLtL2 準拠の Lisp 処理系 Tachyon Common Lisp について報告する。Tachyon Common Lisp は CLtL2 で定義されているすべての機能を含むフルセットの Common Lisp である。

Tachyon Common Lisp の設計方針は

- 1 実行速度が高速であること
- 2 移植性が高いこと

である。そのために核言語はアセンブラで記述し、その他の部分は S 式で記述した。

核言語以外の大部分を S 式で記述したため、インタプリタの移植を行うときは核言語の部分の移植すればよい。

Lisp のアプリケーションでは大きなメモリ空間を使用する場合が多く、またアプリケーションごとにメモリの使われ方は大きく異なる傾向がある。そこでユーザが大きなメモリ空間が使用でき、かつ自由にメモリ空間をカスタマイズ可能にすることにより、効率のよいメモリ使用ができることを目標にした。Tachyon Common Lisp ではコンス、シンボルは最大 4 Gbyte、その他のタイプは最大 256 ~ 512 Mbyte の空間が使用でき、メモリ空間を分割して管理するようにして、実使用空間の動的な変更を可能にした。

大きなメモリ空間を使用すると問題となるのは GC 時間と GC で使用されるメモリ空間である。複

写方式の GC の場合、ヒープ空間として 2 倍の大きさが必要なため、特に大きなメモリ空間を複写するときはメモリ効率を悪くする。Tachyon Common Lisp では圧縮方式の GC を採用している。また圧縮するメモリ空間を分割し、GC 対象にするメモリ空間を制御することにより短時間でできる GC も可能にした。

Tachyon Common Lisp は現在 i860TM RISC CPU を持つワークステーション上で動作する。

また Tachyon Common Lisp では、CLtL2 の拡張機能として以下のものを用意している。

- 1 日本語文字のサポート
- 2 他言語とのインタフェース[7]
- 3 UNIX インタフェース
- 4 CLX インタフェース
- 5 Emacs インタフェース
- 6 デバッグなどのプログラム開発環境

2 Tachyon Common Lisp の構成

Tachyon Common Lisp の構成を図 1 に示す。Tachyon Common Lisp の核言語 PLisp (Primitive Lisp) は一部を C 言語で記述した以外はアセンブラで記述した。アセンブラで記述することにより、実行速度の高速化及びコード領域の縮小が行えた。また、頻繁に使用するデータをレジスタに専用に割り当てて高速化が行える。レジスタ割り当てについては後で述べる。

算術演算ライブラリ、メモリ獲得・解放、ファイルシステムなどの OS インタフェース部分と、それほど速度を要求されない部分は移植性を高めるために C 言語で記述した。

その他の Common Lisp 関数は PLisp で記述し、巡回的に実現した。コンパイラやデバッグ、プリティプリンタなどは Common Lisp 関数で記述した。

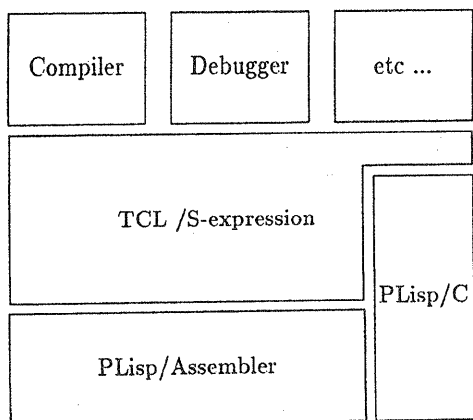


図 1: Tachyon Common Lisp の構成

2.1 移植性

アセンブラで記述されたプログラムは、別の CPU を搭載しているマシンに移植するときには、大幅にコードを書き換えなければならない。

そこで我々はその負荷をなるべく軽減できるように強力なマクロ機構を持つアセンブラ処理系[4]を新たに作成した。この新しいアセンブラは Lisp に似た構文規則を持ち、Common Lisp の持つ強力なマクロ機構と同等なマクロ機構を持つ。例えば RISC CPU の持つ特徴の一つに遅延分岐があるが、この分岐機能をマクロにして隠蔽することにより開発者に遅延分岐を意識させることなくプログラムを記述できるようにした。CPU 依存部の大部分をマクロで記述して、マシン依存のコードをプログラムから隠蔽している。移植を行うときはマクロを変更することで移植のほとんどが行われる。

PLisp 以外の Common Lisp 関数は S 式で記述されている。S 式で実現されている部分は 3 つのレベルに分けられる。最初のレベルは PLisp のみで記述され、次のレベルは PLisp に最初のレベルで実現されたいくつかの基本的な機能が加わった Common Lisp サブセットで記述されている部分である。最後のレベルは Common Lisp フルセットで記述された部分である。移植の時に PLisp の仕

様に変更がなければ、この S 式で記述された部分はそのまま使用でき、PLisp の仕様に変更があった場合でもレベルに応じた変更のみで移植が行えるようにしている。

表 1: S 式レベル

レベル	実現言語	実現機構
レベル 1	PLisp のみで記述	原始関数定義、原始入出力など
レベル 2	PLisp とレベル 1 のみで記述	コンパイルコード読み込み機構、代入機構、パッケージなど
レベル 3	フル Common Lisp で記述	その他

コンパイラ[3]ではフェーズを設け、マシン依存の部分と非依存の部分に分け、依存部を表駆動にすることにより、移植性を高めている。

2.2 核言語 PLisp

Tachyon Common Lisp の核言語 PLisp は初期イメージ部、メモリマネージャ、評価機構、インタプリタ関数部、コンパイラエントリ部、UNIX/C インタフェース部からなり、アセンブラおよび一部 C 言語で記述されている。

初期イメージ部はリスブオブジェクトの初期配置状態をプログラムした部分である。この配置状態を直接アセンブラまたは C で記述するのは困難であるので、メモリ初期状態の記述用言語 InitLisp を作成し、その上でリスブライクな表記でシンボル定義、S 式関数定義をコンス、ベクタ、固定整数、パッケージ、ハッシュテーブルのタイプを使用して初期のメモリ配置状態を記述した。後で述べるが、この部分はガーベジにならないので GC の圧縮対象にしない特別な領域としてメモリ管理を行っている。

Common Lisp では多数のタイプが存在し、いくつかのタイプはフィルポインタ、共有、大きさの変更など高い機能を持っている。また Common Lisp ではそれらの機能がない単純なタイプも用意している。しかしそれらの単純なタイプは高機能なタイプの一部として実現して、その単純なタイプを特別に実現しなくてもよい。Tachyon Common Lisp では単純なタイプを高機能なタイプとは別に実現

した。

すなわち、コンス、固定整数、単純文字列、単純ベクタに対して有用であると思われるタイプスペシフィックな関数を PLisp として実現した。その結果 PLisp はコンパクトなものになった。

3 データ構造／レジスタ

本章では Lisp オブジェクトの構造とレジスタ割り当てについて述べる。

3.1 リスプオブジェクト

Tachyon Common Lisp ではタグ部を持つ 32 ビットの Lisp オブジェクトを採用した。タグは 3～8 ビットの可変長で下位部に配置し、最下位ビットを GC ビットとして使用している。タグは 20 種類あり、その一部を表 2 に示す。

表 2: タグ(一部)

タイプ	タグ
シンボル	010
コンス	110
固定整数	000000
単純ベクタ	001100
単純文字列	000100
関数	100000
シングルフロート	010000
文字	10111100

固定整数のタグは全て 0 であり下位部にタグがあるため固定整数の加減算は 1 ステップで実行される。また固定整数の演算で結果が無限長整数になるときは整数オーバーフローのフォールト機能を使って実現しているため、プログラムによるチェックは必要ではない。

Tachyon Common Lisp では、シンボル、ダブルフロート、コンスなどは 8 バイト境界、シングルフロート、単純ベクタ、単純文字列などは 4 バイト境界でメモリを確保するため、アドレスの下位 2～3 ビットは 0 であることを保証している。

コンス、シンボルは下位 3 ビットが 0 でタグも 3 ビットであるのでアドレスとタグをそのまま加えてリスプオブジェクトにする。これらのタイプに

おいてアクセスは 1 ステップでできる。6 ビットタグのタイプでは 0 となる下位 2～3 ビットを省いてタグに加えることにより、アクセスできるメモリ空間を最大 256～512 MByte までサポートしている。これらのタイプのアクセスも 2 ステップのみ必要とするだけである。これらのタイプに対するデータ参照の例を以下に示す。

```
# コンス(r16)の CAR 部を取り出す
ld.l -6(r16), r17
# コンス(r16)の CDR 部を取り出す
ld.l -2(r16), r17
# 単純ベクタ(r16)の第 1 要素を取り出す
shr 4, r16, r17
ld.l 0(r17), r18
```

3.2 単純ベクタのデータ構造

単純ベクタの例を図 2 に示す。矢印が Lisp オブ

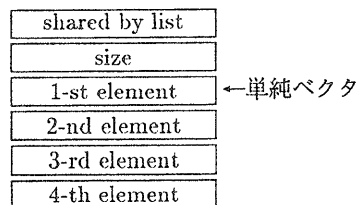


図 2: 単純ベクタのデータ構造

ジェクトのポインタ部に入っている値である。単純ベクタの n 番目の値を取り出すときは以下のステップで実行できる。但し、この例ではタイプチェックも配列の大きさのチェックも行っていない。

```
# ベクタ(r16)の  $n$  番目(r17)を取り出す
shr 4, r17, r17
shr 4, r16, r16
ld.l r17(r16), r18
```

配列、文字列、単純文字列なども同様な構造をしている。

3.3 レジスタ割り当て

i860™ は整数レジスタ($r0 \sim r31$)、浮動小数レジスタ($f0 \sim f31$)を各々 32 個持っている。Tachyon Common Lisp はアセンブラで記述しているため、整数レジスタを表 3 に示すように専用で割り当て

ている。浮動小数レジスタは定数の格納や、最大 128 ビットのデータを扱えるためメモリ領域のブロック転送などのワークレジスタとして使用している。

表 3: レジスタ割り当て

レジスタ	名称、機能
r0	ALL ZERO
r1	リターンアドレス
r2	システムスタックポインタ
r3	リスブフレームポインタ
r4	リスブスタックポインタ
r5	環境フレームポインタ
r6	システムモード
r7	グローバルポインタ
r8	NIL
r14	レスト引数の格納アドレス
r15	引数の個数/値の個数
r16 ~ r23	引数/値
その他	ワーク用

リスブスタックポインタをレジスタに専用に割り当てることにより、頻繁に起こるスタックアクセスが 1 ステップで実行される。またスタックオーバーフローのチェックは UNIX のページフォールトの機能を使って実現しているので、プログラムによるチェックは必要でない。

システムモードは実行に大きく影響を与える変数 `*evalhook*`、`*applyhook*` の値の管理や局所関数の管理などに使用する。例えば、`*evalhook*`、`*applyhook*` などの変数の値が Common Lisp の既定値であるときは、システムモードの対応するビットを 0 にしている。このシステムモードを調べるだけでそれらの変数の値が既定値かどうかの判断ができる。

グローバルポインタはレジスタに割り当てられていないデータのアクセスを高速にするために使用している。これは i860™ が 1 ステップで与えられたメモリアドレスからデータを読みこむことができないため、グローバルポインタにそれらのデータの先頭アドレスを格納しておくことにより 1 ステップでデータを読み込めるようにした。その例を以下に示す。

通常のメモリ読み込み

```
orh  address@ha, r0, r31
ld.l address@l(r31), r31
# グローバルポインタを用いる
ld.l offset(r7), r31
```

レジスタ r16 ~ r23 は引数の受け渡しに使用される。PLisp 関数やコンパイルコードでは 8 個までの引数はスタックではなくレジスタ r16 ~ r23 に格納し、それ以上の引数はスタックに格納し、r14 にそのスタックアドレスを設定する。

PLisp 関数では引数の個数は 8 個までに制限したので引数はレジスタ渡しになる。また PLisp 関数のレスト引数はリストにして渡すのではなく、8 個までならレジスタに、それ以上ならスタックに格納して受け渡す。

多値の扱いも引数のときと同様で、値の個数が 8 個までのときはレジスタ r16 ~ r23 に格納し、8 個より多いときは静的な多値専用領域に格納する。また値の個数を r15 に設定する。PLisp 関数では値の個数を 8 個に制限したので値はすべてレジスタ渡しになる。

以上より高速なデータのアクセスができるようになった。

4 メモリマネージャ

Lisp ではメモリ管理は処理系が行うため、ユーザはメモリを気にせずに使えという利点がある。Lisp のアプリケーションプログラムは大量のメモリを消費する場合が多く、その使用状況もアプリケーションで大幅に異なってくる。そのため自由にメモリの使用の設定を変更できることが望ましい。

Tachyon Common Lisp ではコンス、ベクタ、シンボル、文字列、コンパイルコード、シングルフロート、ダブルフロートなどのメモリ種別に分け、専用の領域を確保しそれぞれの領域を分割して管理する方法を採用している。

領域は関数 GET-MEMORY を実行することにより動的に確保される。同じメモリ種別を引数にして、関数 GET-MEMORY を複数実行することによって、一つのメモリ種別に対して複数の領域が確保される。また、GC のモードをフリーモードにすることにより未使用の領域は動的に解放される。メモリ領域の例を図 3 に示す。FreePtr は現在の

オブジェクトの生成する場所を指している。

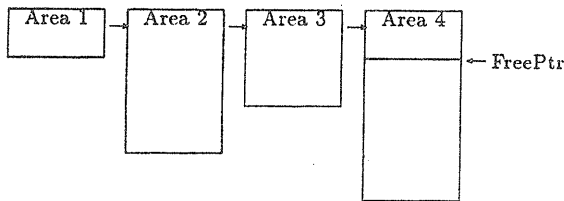


図 3: メモリ領域の例

またアプリケーションを動作させ、メモリが足りなくなると自動的に領域を確保する。

このような一つのメモリ種別に対する複数の領域を、次の節で述べるエフェメラルGCにおいて、圧縮対象の領域を制御する単位に使用する。

以上のようなメモリ管理方式により、アプリケーションに適したメモリの使用ができる。

4.1 エフェメラルGC

複写方式のGCを用いるとヒープ領域が2倍必要になり、大きなメモリ空間を使用するアプリケーションの場合メモリ効率が悪い。そこで Tachyon Common Lisp では圧縮方式のGCを行っている。

圧縮方式のGCは生きているオブジェクトのマーキング、圧縮表の作成/圧縮、アドレス変換のフェーズから構成される。アドレス変換を行うときに圧縮幅を圧縮表から探索するが、Tachyon Common Lisp では2分探索と順次探索の組み合わせで行っている。両者の探索ステップ数が、2のべき乗の数としては16で逆転するためである。

圧縮は各メモリ種別単位に行われる。一つのメモリ種別が複数の領域を持つため、圧縮対象にする領域をこれらの領域に対して制御することにより、圧縮に要する時間とガーベジの回収量を制御する。また、初期イメージによって確保されているシステムメモリ領域はガーベジにならないので、この領域に対して圧縮は行わない。以下に出てくる領域数についてもこの領域を含んでいない。

ある種別の領域が n 個存在し、そのうち m 個の領域が使用されているときにはGCのモードにより以下の領域を圧縮する。但し、あるメモリ種別においてメモリを確保した順に領域番号を1, 2, 3, ...のように領域に与える。領域番号 i で示される領域を領域 i と呼ぶ。領域 $i+1$ は領域 i の次に確

保された領域である。領域 m はフリーポインタがある領域、即ち現在オブジェクトが生成されている領域である。また g はエフェメラルGCが対象とする領域の個数であり、その初期値は1である。

1. ダイナミックGC

領域1から領域 m までを領域1方向へ圧縮する。

2. エフェメラルGC

領域 $m - g + 1$ から領域 m までを領域 $m - g + 1$ 方向へ圧縮する。

本方式の圧縮型GCは領域番号の小さい領域の方向へ圧縮をするので、小さい領域番号の領域に古いオブジェクトが集まり、またそれらはガーベジになりにくい性質がある。逆に領域 m やそれに近い領域は新しいオブジェクトの集まりであり、ガーベジになりやすい性質がある。エフェメラルGCはそのガーベジになりやすい領域のみを圧縮することにより、GCの効率を向上させている。ここでGCの効率とは、ガーベジの平均回収量と平均GC時間の関数で表される。

Tachyon Common Lisp のマークフェーズではすべてのオブジェクトを対象にしている。マークする対象も古いオブジェクトから新しいオブジェクトを指すものを別に管理することにより、エフェメラル領域だけにしてマークを行うことが可能である。しかしデータアクセスを高速に行うために古いオブジェクトから新しいオブジェクトを指すデータを管理するためのステップをなくすことと、GCビットがGC時以外のときは0である保証をする必要があることと、さらに圧縮フェーズが一番の負荷になっていることから、圧縮する領域のみを制御するGCを採用した。

g を制御することによりGC対象の領域を動的に制御できる。Tachyon Common Lisp における g の変更アルゴリズムは以下のとおりである。

1. オブジェクトの回収量が下限閾値 L より小さいときは、 $g \leq m$ の範囲で g を1つ増加させる。
2. オブジェクトの回収量が上限閾値 U より大きいときは g を1にする。

g はなるべく小さい方がGCに要する時間が少

ない。またガーベジを大量に生成するときは g は 1 のまま変動しない。

g の増加分 Δg を 1 に固定すると、GC が必要以上に小さな時間間隔で起きることがある。

4.2 回収量の推測による GC

Δg を定数 1 とせずに、ガーベジの回収量を GC を行う前にある程度推測できるならば、次の GC に適した g を選択でき、効率のよい GC を行えるであろう。

推測のための解析については別の機会に報告するが、ここでは回収量の推測結果を用いて圧縮領域を制御する GC 方法について述べる。

まず、領域を 3 個のグループに分ける。最初のグループは前回の GC 後のフリーポインタから現在の GC が起動する時点でのフリーポインタまでの領域であり、次のグループは最初のグループを除いた、領域の中のオブジェクトが一度も圧縮されていない領域である。最後のグループは一度以上圧縮された領域である。推測したガーベジ分布関数からそれぞれのグループの回収量を計算して、その値の合計を G_p とする。

$L < G_p$ になるまで Δg の値を増加させる。 Δg を 1 に固定的にするよりも、平均回収量が大きくなり、平均 GC 時間は回収量が距離(フリーポインタからのオブジェクト数) に対して負の相関を持っているためそれほど大きくならず、効率のよい GC ができる。

このようにして g を決定する GC のシミュレーションプログラムを作成した。このシミュレーションプログラムはシミュレーション用言語 `smpl` [8] で記述した。`smpl` は C 言語上のライブラリとして動作するが、我々はそれを Common Lisp に移植し Tachyon Common Lisp 上で動作させた。 Δg を固定にする方式の測定結果との比較を後で述べる。

4.3 その他の GC モード

エフェメラル GC とは別に以下の GC のモードがある。

1. フリーモード

領域 $m+1$ から領域 n までをフリーにする。

2. ノー GC モード

GC を行わずにメモリ獲得のみを行う。

また圧縮対象のメモリ種別は、関数 GC によって GC が起動されたときはすべてのメモリ種別を対象にする。生成操作関数によって、すべてのメモリを消費したときに呼び出されるエフェメラル GC では GC 要因となったメモリ種別だけを GC の圧縮対象にする。また、GC モードによりすべてのメモリ種別を対象にすることもできる。

5 性能評価

Tachyon Common Lisp は現在 i860TM RISC CPU (40MHz) を持つワークステーションで動作する。Gabriel ベンチマークプログラムのインタプリタの実行速度を表 4 に示す。

核言語 PLisp の大部分をアセンブラで記述したことや各種の高速化を行ったことにより、ベンチマーク結果の示すとおり、インタプリタの基本機能は非常に高速となっている。

表 4: ベンチマーク(インタプリタ)

Gabriel Benchmark	Interpreter
tak	1.589
stak	3.031
ctak	2.317
takl	12.671
Takr	1.918
Boyer	26.600
Browse	39.910
Destruct	7.350
Init-traverse	53.700
Run-Traversal	267.400
Derivative	4.090
Dderivative	4.940
Fft	2.170
Puzzle	40.340
Triangle	480.370

単位は秒

GC の計測時間として、全メモリ領域が約 6 メガバイトでコンスメモリ領域を 10 分割にして、S

式で記述したシステムの立ち上がり直後のGCにおけるCPU時間を表5に示す。但し g は1である。

表 5: GC時間

GC 種別	対象	時間(sec)
Dynamic GC	All Memory	2.910
Ephemeral GC	All Memory	1.500
Ephemeral GC	Cons Memory	1.290

メモリを大量消費していれば、ダイナミックGCとエフェメラルGCの差は大きくなる。S式システムの立ち上がり直後では多くのコンス領域を使用してまたガーベジも多いので、この状態でGCの評価を行った。一般のアプリケーションで大量にコンス領域を消費する場合も同様の結果が得られるであろう。

シミュレーションモデルでは領域を15分割にして、それぞれのサイズを100にした。下限閾値 L を平均領域サイズの20%である20、上限閾値 U を50%の50にした。ガーベジの生成される確率を70%にし、その寿命は平均10回の生成操作が行われる時間の指数分布に従うようにした。エフェメラルGCを20回行うとダイナミックGCを行うようにした。

シミュレーションの結果を表5に示す。平均GC時間は平均圧縮領域サイズと正の相関がある。

この表から分るように Δg を固定にするのではなく、回収量にある程度予測して、それに基づいて g を変化させる方が平均回収量は大きく、平均圧縮領域サイズはあまり変わらない。

6 おわりに

CLtL2仕様に基づく高速でかつ高い移植性を持つCommon Lisp処理系であるTachyon Common Lispの実現方式について報告した。Tachyon

表 6: GCシミュレーションの結果

	Δg 固定	Δg 可変	dynamic
1を越える Δg の回数	0	3.4	0
GC回数	69.8	66.2	19.2
平均圧縮領域サイズ	243	243	1505
平均回収量	50.8	53.6	181.4

Common Lispはアセンブラで核言語を記述したため、専用にレジスタを割り当てることができ、i860TM CPU特有の最適化を行い、高速な実行速度を確認できた。

移植については核言語以外は可搬性でレベル分けされたS式で記述し、核言語の記述においても高機能なマクロ機能を持つアセンブラを作成し、マシン依存部を隠蔽している。今後、他のCPUを持つマシンに移植して高い移植性の確認を行う。

Tachyon Common Lispのメモリマネージャは大容量のメモリをサポートし、ユーザカスタマイズが行いやすく、さらにGCは短時間で終了するようにGC対象領域を制御できる。

Tachyon Common Lispは現在OKIstation 7300上のOKI Common Lispとして製品化されている。

今後の課題としてウインドウインタフェースを含む開発環境の整備があり、現在開発中である。

参考文献

- [1] Guy L. Steele Jr.: "COMMON LISP THE LANGUAGE SECOND EDITION", Digital Press, 1990.
- [2] Intel Corporation: "i860TM 64-BIT MICRO-PROCESSOR PROGRAMMER'S REFERENCE MANUAL", Intel Corporation, 1989.
- [3] 新谷他: "OKI Common Lispの開発ーコンパイラー", 情報処理学会第43回全国大会 5L-4
- [4] 高橋他: "OKI Common Lispにおけるアセンブラプログラムの開発環境", 情報処理学会第43回全国大会 1J-7
- [5] 山田他: "OKI Common Lisp インタプリタの実現", 情報処理学会第43回全国大会 5L-3
- [6] 大江他: "OKI Common Lispの開発ー概要ー", 情報処理学会第43回全国大会 5L-2
- [7] 山崎他: "Common Lispにおける他言語インタフェース" 情報処理学会第43回全国大会 5L-6
- [8] M. H. MacDougall: "SIMULATING COMPUTER SYSTEM: Techniques and Tools", MIT Press, 1987