愛知教育大学研究報告，53（自然科学編），pp. 27～32, March, 2004

Common Lisp 言語処理系の64ビット化

安本太一

情報教育講座

A Common Lisp System on 64 Bit Environment

Taichi YASUMOTO

Department of Information Sciences, Aichi University of Education, Kariya 448-8542, Japan

1．は　じ　め　に

さまざまなアーキテクチャの64ビットCPUが発表され提供されている。64ビットCPUの能力を生かすためのオペレーティングシステムや，64ビットCPUの能力を生かしたアプリケーションも開発され提供されてきている［1，2，3］。

64ビットCPUそのものは10年以上も前に発表されていたが，その当時は64ビット環境が普及する環境が整っていなかった。OS（オペレーティングシステム）は64ビットCPUに暫定的に対応したというような状況で大規模ファイルの扱いが十分に整備されていなかったり，ハードウェア（大容量のメモリやハードディスク，64ビットCPUそのもの）も高価であった。しかし，本稿を執筆している時点では，SPARC系の64ビットCPUを搭載したワークステーションやPowerPC系の64ビットCPUを搭載したパーソナルコンピュータが20万円前後で入手可能であり，64ビット環境が身近なものとなってきている。これらのコンピュータに4GB以上のメモリを搭載することも，現実的な費用で可能である。データマイニング，大規模データベース，マルチメディアといった大容量のメモリを必要とするような処理が行なわれる機会も多くなり，アプリケーションの64ビット化への期待は一層高まっている。

アプリケーションの開発には，プログラミング言語処理系が64ビット環境に対応することが必要である。本稿では，KCL（Kyoto Common Lisp）［4，5］という従来32ビット環境で動作していたCommon Lisp言語処理系を64ビット環境へ移植したときの経過を報告する。

2．64ビット環境

本稿の対象としている64ビット環境は，従来のKCLが動作の対象としていた32ビット環境と比べて，次のような変更点がある。

2.1　データ型のサイズの変更

データ型のサイズに関して，本稿の対象としている64ビット環境をLP64という。従来の32ビット環境をILP32という。KCLの記述言語でもあるC言語のデータ型で述べると，LP64では，long（長い整数）が32ビットから64ビットへ，ポインタが32ビットから64ビットへと変わる。char（1バイトの文字）が8バイト，short（短い整数）が16ビット，int（普通の整数）が32ビット，float（単精度浮動小数点数）が32ビット，double（倍精度浮動小数点数）が64ビットであるというのは，ILP32のときと変わりない。

2.2　アラインメントの変更

たいていの64ビットCPUでは，データの配置方法，すなわちアラインメントの制約は次のように変わる。

1．8バイトのデータは，8の倍数のアドレスに配置される。longやポインタは32ビット環境下では4の倍数のアドレスに配置されるが，64ビット環境下では8の倍数のアドレスに配置される。この配置は，longやポインタが構造体のメンバとなっているときも行なわれる。

2．32ビット環境では，配列データやユーザが動的に獲得するメモリ領域の先頭のアドレスは4の倍数であるが，64ビット環境では8の倍数になる。

3．ユーザ定義のデータ型である構造体では，そのデータの先頭アドレスは32ビット環境下では4の倍数となるが，64ビット環境下では8の倍数となる。変数宣言によって静的に構造体データを確保した場合はもちろんのこと，mallocなどによる動的なメモリ割り当てによりあらかじめ大きなメモリ領域を確保しておいて構造体のテンプレートに割り付けていく場合も，このアラインメントに従わなければならない。

このアラインメントの制約は，メモリアクセスの高速化やハードウェアの簡素化を目的とした，CPUの

ハードウェア構成に起因するものである。状況によってはメモリの未使用領域が生じるためメモリ効率を低下させる。

2.3 オブジェクトコードのサイズの増加

絶対アドレス，相対アドレスのオフセット，即値（longの即値）が32ビットから64ビットになるので，これらの数値の分だけコードサイズが増加する。この数値の増加は，RISCアーキテクチャでは，命令数の増加につながる。RISCでは，命令セットを簡略化し命令長を（比較的短い）固定長にしている。それゆえ，32ビット環境のときでさえも，1命令で32ビットの数値をレジスタにロードするのではなく，複数の命令で分担して数値のロードを行なっている。64ビットの数値ともなれば，32ビットの場合よりさらに命令が必要で，字面上のステップ数が増加しコードサイズが増加する。ただし，64ビットCPUともなれば命令のパイプライン実行や並列実行などの高速化機能が備わっているであろうから，必ずしも実行時間の増加につながるものではない。

3．64ビット環境へLisp言語処理系を移植する利点と必要性

64ビット環境へLisp言語処理系を移植することについて，次のような利点と必要性がある。32ビット環境と比較するとき，メモリ空間の拡張や整数演算の高速化について言及されることが多いが，それだけではない。

1．膨大なアドレス空間の獲得
Lisp言語は記号処理を得意とする言語ではあるが，Lisp言語とその処理系はその特徴からプログラムの生産性が高いといわれ，さまざまな分野で用いれらている。データマイニングといったような大量のメモリを必要とする分野のプログラムをLisp言語で記述したいといった要求も当然でてくるであろうから，Lisp言語処理系そのものを64ビット環境に対応させる必要がある。

2．整数演算の高速化
Common Lispでは無限長整数（bignum）というのがあって，メモリが許す限り，大きな桁数の整数の演算ができる。無限長整数の演算は，従来32ビット毎の演算（とキャリーやボローの伝搬）に分割されて行なわれていた。これを2倍のビット長の64ビット毎に行なうことにより，演算（とキャリーやボロー伝搬）の回数がほぼ半減することから，無限長整数の演算の高速化が期待できる。
固定長整数（fixnum）で表現できる範囲が32ビットから64ビットで表現できる整数まで広がることから，無限長整数を使用せずにすむケースが増えるので，整数演算全体の高速化に寄与する。

3．他言語で開発されたプログラムとのリンク
プログラミング言語がそれぞれ得意とするところを生かし，大きなプログラムを複数のプログラミング言語で記述することがある。そのようなことを可能にするために，処理系そのものの64ビット化が必要である。
Lisp言語処理系のいくつかは，Fortranなどの他の高級言語と同様に，C言語プログラムとのインタフェースをもっている。ただし，Sun Microsystems社のSolarisにみられるように，OSが用意しているリンカは，32ビット環境を想定してコンパイルして作成したオブジェクトコードと64ビット環境を想定してコンパイルして作成したオブジェクトコードを混在してリンクすることはできない。したがって，Lispの関数から64ビット環境を想定したC言語の関数を呼び出したり，64ビット環境を想定したC言語の関数からLispの関数を呼び出す場合は，Lisp言語処理系そのものが64ビット環境下で動作しなければならない。

4．OSが提供する新機能や機能向上の利用
OSの新しい版において新機能や機能向上が提供されることがあるが，これらの新機能や機能向上は32ビット環境では提供されず64ビット環境から提供されることがある。例えば，Sun Microsystems社のOSであるSolarisでは，1つのプロセスにおいて同時に開くことができるファイルの数が大幅に増えるといった機能向上は，64ビット環境でないと利用できない。このような事例から，64ビットCPUによる恩恵（膨大なアドレス空間や整数演算の高速化）とは直接関係ないように思われるところで，64ビット環境への対応が求められることがある。

5．全般的な処理速度の向上
64ビットCPUには，パイプライン，分岐予測の精度向上，並列実行，広帯域バスなどいった高速化のための最新技術が多数採用されている。膨大なアドレス空間の必要性がないプログラムを実行する場合でも，高速化の恩恵が得られる。64ビットCPUの中には，PowerPCのように32ビット環境を前提にした実行オブジェクトをネイティブ実行できるものもある。その一方で，IA-64アーキテクチャであるItaniumのように，32ビット環境の実行オブジェクトをネイティブではサポートせずエミュレーション実行するCPUがあり，Lisp言語処理系そのものを64ビット環境に適応させないと十分なパフォーマンスが得られないものがある。

4．過去の研究

佐藤らにより，DEC社（Compaq社を経て現在はHewlett-Packard社）の64ビットCPUであるAlpha

を搭載したワークステーションを対象にPHL（Portable Hashed Lisp）というLisp言語処理系を移植した報告が行なわれている[6]。しかしながら，この報告は，ごみ集めの報告が中心であり，シェアが小さいAlphaやPHLを対象にしている。観点をかえたり，他のプラットフォームや他のLisp言語処理系を対象にした報告を加えることによって，64ビット環境におけるLisp言語処理系の実装についての技術的なデータを充実させることが重要である。本稿では，64ビットCPUであるUltra SPARC IIiを搭載しOSとしてSolarisを使用しているワークステーションにおいて，KCLというCommon Lisp言語処理系の64ビット環境への移植について，メモリ管理に重点をおいて報告する。

5．64ビット環境への移植

今回行なったKCLの64ビット環境への移植について，主要な点を述べる。

5.1 データ型についての基本方針

KCLのシステムはC言語とLisp言語によって記述されている。Common Lispの各データ型はC言語の構造体を使って表現され，データの実体であるオブジェクトへのポインタは，さまざまなデータ型を総称する共用体へのポインタとして実現されている。

この構造体中のメンバのうち，ポインタとなっている部分を64ビットにした。64ビットCPUのアドレス空間のどこでも指すことができるようにするためである。一方，先述したPHLという処理系では，64ビットのポインタの上位3ビットをデータ型を示すタグとして使用している。上位3ビットを使用するような高位のアドレスを使うことはまずないという判断をされたのであろうが，メモリの割当てが低位のアドレスから行なわれる保証はないのだから，さまざまなCPUやOSへの移植性を考えるとPHLの実装は必ずしも適当ではない。

KCLにおいてデータ型を表現している構造体において，ポインタとなっているメンバは，陽にLispのデータを指しているものと，内部的に他のデータを指しているものにわけられる。前者は，例えばコンスデータのcar部とcdr部というようにそのデータ型の定義そのもののとおりLispのデータオブジェクトを指している。後者は，一次元の配列であるベクタのような可変長のオブジェクトのヘッダ（C言語の構造体で定義）から本体（他のオブジェクトへのポインタを複数個格納するための連続したメモリ領域）を指すというような内部的なもので，KCLの利用者からは陽にみえないものである。もちろん，このような陽にはみえないがLispのデータあるいは内部的なデータを指す部分も，64ビットにした。

ポインタを64ビットにするのはコンパイラのオプションに64ビットモードを指定するだけである。KCLのソースコードに多大な修正を要するのは，これらのメンバにアクセスしているコードを64ビット環境に対応させることである。例えば，ポインタをintにキャストしているところ，ポインタをintの変数に代入しているところ，関数の引数としてポインタを受けとるのに引数の型宣言がintであるといったところは，ポインタの上位32ビットが落ちてしまうのでlongを使うように修正する。

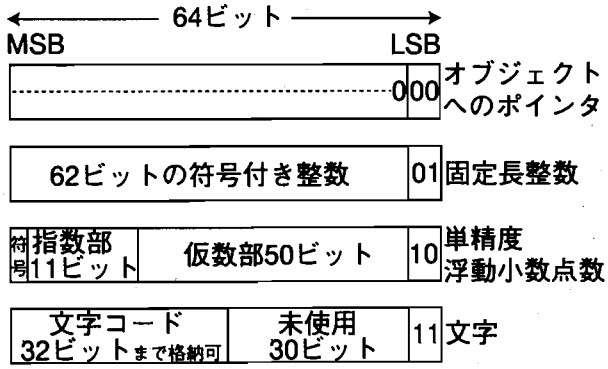

図1 64ビット環境下のオブジェクトの表現

5.2 即値データ

32ビット環境下のときから即値データとして実現されているデータ型である固定長整数(fixnum)，単精度浮動小数点数(short-float)，文字(character)については，64ビットCPUの能力を生かすように実装を変更した。そのフォーマットを図1に示す。Lispのデータオブジェクトは，通常そのオブジェクトへのポインタで参照されるが，ポインタ内に直接オブジェクトをコーディングする方法があり，このようにコーディングされたオブジェクトを即値データという[7]。即値データには，オブジェクトへの参照が高速になる，メモリを消費しないといった利点がある。

先述の8バイトのデータのアラインメントの制限により，オブジェクトへのポインタは必ず再下位から3ビット目まで必ず0になる。そこで，下位3ビットが000以外の場合に，オブジェクトをポインタ内にコーディングすることができる。64ビット環境では，固定長整数は30ビットから62ビットへ，単精度浮動小数点数は30ビットから62ビットへ，文字は16ビットから32ビットへ，それぞれ32ビット環境のときからビット長を増やした。

単精度浮動小数点数(short-float)が62ビットというのは，64ビットである倍精度浮動小数点数（double-float）とほとんど同じではないかとう異論があるかもしれない。事実，この単精度浮動小数点数(short-float)の表現における演算は64ビットで行ない（C言語でいうとdoubleで演算を行ない），その結果の仮数部の下位2ビットを削っているだけである。最近のCPUは，

単精度浮動小数点数（C 言語の float）も倍精度浮動小数点数（C 言語の double）と同様に内部的には64ビットで演算を行なっていることを考えれば，short-float を62ビットで表現しても実行効率の点では不都合はない。しかも，Common Lisp の言語仕様［8］では，単精度と倍精度のビット長の大小関係を定義しているだけで，具体的なビット長は定義していない。もちろん，short-float を32ビット表現にして実装することも可能であり，その場合でも32ビット環境のときより仮数部が2ビット増えるので精度があがる。

文字は，文字コードのための領域を32ビットとしたので，将来4バイト表現のユニコードに対応することができる。現在は，この領域の16ビット分だけを使い，32ビット環境下のときと同様に1バイトの英数字のみならず2バイトの日本語文字にも対応している。

5.3 アラインメント

32ビット環境から64ビット環境におけるアラインメントの変化は，メモリ効率の低下を招いた。例えば，図2に示すとおり，32ビット環境下ではコンスセル1つの大きさは12バイトだったが，64ビット環境下では12バイト増えて24バイトになった。ポインタである car 部と cdr 部がそれぞれ4バイト増加して合計20バイト（12バイトに8バイト増）となることが期待されるが，8バイトのデータは8の倍数のアドレスに配置するというアラインメントの制限により，4バイトの未使用領域が生じている。

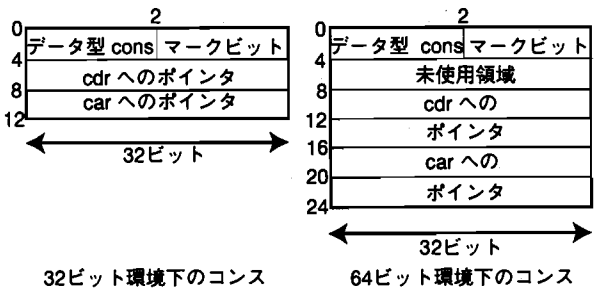

図2 コンスセルの構造

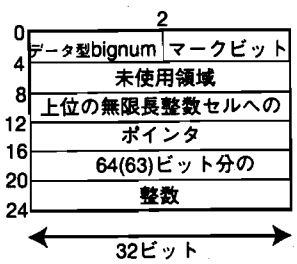

図3 64ビット環境下の無限長整数セルの構造

5.4 無限長整数

無限長整数（bignum）の表現を図3に示すように変更し，加減乗除などの演算を（32ビット毎だったものを）64ビット毎にできるようにした。内部的な加減乗除などの演算の回数がおおよそ半減することになり，実行効率が向上することが期待される。

コンスセルの場合と同様に4バイトの未使用領域が生じている。これが好ましくないならば，この領域にも無限長整数を構成するビットを格納するという手法も考えられ，無限長整数の演算を32ビット毎の演算と64ビット毎の演算を組み合わせて実現した場合の実行効率とあわせて検討する価値がある。

5.5 データサイズやアラインメントの変化と最適化

KCL では実行効率やメモリ効率を向上させるために，データ表現に工夫をしている。この工夫が，ポインタのデータサイズの変化やアラインメントの変化の影響を大きく受けた。例をあげて説明する。

いくつかの関数では，名前を示すものとして使う場合は，記号（symbol）と文字列（string）のどちらでも引数としてとることができる。これらの関数の内部では，引数が記号あるいは文字列のいずれかであることを一度チェックした後は，記号と文字列の区別なく全く同じコードが実行される。これを可能にしているのは，次に記号と文字列の定義を示すように，記号の印字名を指し示す s_self というメンバと，文字列の本体を指し示す st_self というメンバが，双方とも先頭からのオフセットが（32ビット環境下では）同じ16バイトになっているからである。そして，記号の印字名 s_self を，文字列の構造体テンプレートを使ってアクセスしたりすることができる。

```
/* 32ビット環境下の記号（symbol）の定義 */
struct symbol {
  short t, m; /* type tag and mark bit */
  object s_dbind;     /* dynamic binding */
  int     (*s_sfdef)() ; /* special form */
#define NOT_SPECIAL     ((int (*)()) Cnil)
#define s_fillp       st_fillp
#define s_self        st_self
  int       s_fillp; /* print name length */
  char    *s_self; /* print name */
  object  s_gfdef; /* global function */
  object  s_plist; /* property list */
  object s_hpack; /* home package */
  short   s_stype; /* symbol type */
  short   s_mflag; /* macro flag */
};

/* 32ビット環境下の文字列（string）の定義 */
struct string {
  short t, m; /* type tag and mark bit */
  short st_hasfillp;     /* fill-pointer flag */
  short st_adjustable; /* adjustable flag */
  int     st_dim;         /* dimension */
```

```
  int     st_fillp;        /* body length */
  char    *st_self;        /* body */
  object st_displaced;     /* displaced */
#ifdef KANJI
  short st_strtype;        /* string type */
#endif
};
```

同様の事例として，ごみ集めの効率化のために，文字列のメンバである st_displaced は，多次元配列(array)の a_displaced や一次元配列（vector）の v_displaced とあわせて，先頭からのオフセットが（32ビット環境下では）20バイトになるように定義されているというのがある。ここでは，多次元配列や一次元配列の定義は示していないが，displaced が後置されたメンバは，他の文字列や配列を参照する機能に関連したメンバという共通点がある。このように，効率の向上のために，異なるデータ型の間で，関連のあるメンバはオフセットを一致させているところがいくつかある。

64ビット環境下では，long やポインタが64ビットになり，int とポインタがともに同じ大きさであるという32ビット環境下での前提がくずれるので，上記で述べたメンバのオフセットの一致がくずれる。さらに，long やポインタは8の倍数のアドレスにのみ配置可能であるというアラインメントの制限が，32ビット環境下の4の倍数の制限に加わり，オフセットの一致を左右する。そこで，記号，文字列，多次元配列，一次元配列のデータ型を定義している構造体のメンバの順序の入れ換えをし，上記で述べたオフセットの一致をさせることができた。ここでは，紙面の都合で，記号についてのみ変更後の定義を次に示す。

```
/* 64ビット環境下の記号（symbol）の定義 */
struct symbol {
  short  t, m; /* type tag and mark bit */
  short  s_stype; /* symbol type */
  short  s_mflag; /* macro flag */
  object s_dbind;    /* dynamic binding */
  int      (*s_sfdef)() ; /*   special form */
#define  s_fillp      st_fillp
#define  s_self       st_self
  int    s_fillp;     /* print name length */
  char   *s_self;     /* print name */
#define NOT_SPECIAL     ((int (*)()) Cnil)
  object s_gfdef;  /* global function */
  object s_plist;     /*   property list */
  object s_hpack;  /*   home package */
};
```

記号の印字名 s_self のオフセットは，先頭から32バイトとなった。データ型 t とマークビット m の後は，8バイトのデータが続く時は，4バイトの未使用領域が生じてすき間となるが，2バイトのメンバ s_stype と s_mflag を後方から移動してこのすき間をうめた。このように，適当なサイズのメンバがあれば，そのメンバの順序を移動することにより，アラインメントの制限によるメモリ効率の低下を抑制することができる。ただし，オフセットの一致やメモリ効率の低下の抑制を優先させると，字面上のメンバの並びが不自然になり，プログラムの保守の観点からは望ましくない。

5.6 その他

今回の64ビット化は，Sun Microsystems 社の Solaris9 の64ビット環境下で行なったが，32ビット環境下と比べて API の定義が近代的なものになっており，ライブラリやヘッダが32ビット環境下のそれと異なる部分があった。例えば，64ビット環境下ではファイル構造体は単なる long の列挙になり，そのメンバの実装の詳細がユーザに対して隠蔽され，メンバへのアクセスは関数を通して行なうようになっていた［3］。このような部分はコンパイル時にエラーになるので，KCL のソースコードの修正が必要であったが，KCL の移植性を高めるのに貢献した。

また，Solaris9 の64ビット環境下では，プログラムがメモリを要求すると得られるメモリブロックは 100000000_{16} 番地から始まっていた。このメモリ割り当ての状況は，今回の64ビット環境への移植に貢献した。ポインタは32ビットであると仮定してアドレスの計算を int でやっているようなコードでは，上位の32ビットの 1_{16} が落ちてしまう。この上位32ビットの 1_{16} が落ちた状態で，メモリアクセスを行なうとプログラムがすぐ異常終了することから，64ビットクリーンでないコードをみつけだすことが比較的容易であった。

6. ま と め

本稿では，Common Lisp 言語処理系である KCL の32ビット環境から64ビット環境への移植について報告した。筆者のところでは，Sun Microsystems 社製の64ビット SPARC ワークステーション Ultra10 上で，64ビット化した KCL が動作しており，現在その性能評価を実施している。

KCL は，その移植性が高いことから，多くの32ビット CPU 計算機の上で動作してきた実績を持つ。Itanium や PowerPC などの他の64ビット CPU を搭載したさまざまな計算機の64ビット環境で，今回の64ビット化 KCL の動作確認と必要な改良をするとともに，その性能評価を行なうことが今後の課題である。

引 用 文 献

［1］清兼義弘：64ビット UNIX & CDE, 共立出版（1997）.

[2] 池井満：IA-64プロセッサ基本講座，オーム社 (2000).
[3] McDougall, R. and Mauro, J. : *SOLARIS Internals: Core Kernel Architecture*, Prentice Hall (2000).
[4] Yuasa, T. and Hagiya, M. : *Kyoto Common Lisp Report*, Teikoku Insatsu Publishing (1985).
[5] Yuasa, T. : Design and Implementation of Kyoto Common Lisp, *Journal of Information Processing*, Vol. 13, No.3, pp.284-295 (1990).
[6] 佐藤圭史，青木徹，寺島元章：Alpha-chip マシン上のPHL 処理系について，情報処理学会プログラミング研究会研究報告，18-8，pp.57-64 (1989).
[7] 湯淺太一，安本太一：KCL における即値データの実装とその評価，電子情報通信学会春季全国大会講論集，D-357 (1989).
[8] Steele, G. L. Jr. : *Common Lisp the language*, Digital Press (1984).

(平成15年 9 月11日受理)