記　号　処　理　47－5
（１９８８．９．２２）

ｋｏｎｏＣＬ用コンパイラ　ＣＬｉＣ

藤岡　秀樹　　中村　輝雄　　納富　雅人　　山本　昌彦

日立ソフトウェアエンジニアリング株式会社

ｋｏｎｏＣＬは、ＭＣ６８０００系ＵＮＩＸワークステーション上で試作しているフルセットＣｏｍｍｏｎＬｉｓｐ処理系である。本報告では、ｋｏｎｏＣＬ用コンパイラＣＬｉＣと、そのコンパイル コードについて述べる。

ＣＬｉＣは、ＣｏｍｍｏｎＬｉｓｐで記述したファイル単位のコンパイラであり、ＣｏｍｍｏｎＬｉｓｐのソースファイルをコンパイルして、ＵＮＩＸのオブジェクトファイルに変換する。出力したコンパイル コードは、ＭＣ６８０００ＣＰＵのレジスタをフルに利用し、変数の参照、ラムダパラメータの解析や関数呼び出しを高速に実現した。

ＣＬｉＣ

— CommonLisp Compiler for konoCL —

Hideki Fujioka　Teruo Nakamura
Masato Noutomi　Masahiko Yamamoto

Research & Development Department,
Hitachi Software Engineering Co.,Ltd.
6-81, Onoe-machi, Naka-ku, Yokohama 231, Japan

KonoCL is a full set Common Lisp system being implemented on UNIX with MC68000 CPU family. This paper describes CLiC, a compiler for konoCL, and its compile-code.

CLiC, a file-to-file translator, compiles a source file of Common Lisp and generates a relocatable object file. ALL of CLiC are implemented with CommonLisp language itself. With full registers of MC68000 CPU, variable reference, lambda-parameter analysis and function call are executed speedily in compile-code.

1.はじめに

アプリケーションを最適な環境で実行できるCommonLisp処理系konoCL(kono CommonLisp)[1]のコンパイラCLiCについて報告する。konoCLは、筆者らが、1985年より独自に試作しているCommonLisp処理系であり、現在 68000系UNIXマシン(HP/9000, SUN3)上で稼動している。これを使って、自然言語処理の研究[2]などを行っている。

konoCLの構成を図１に示す。

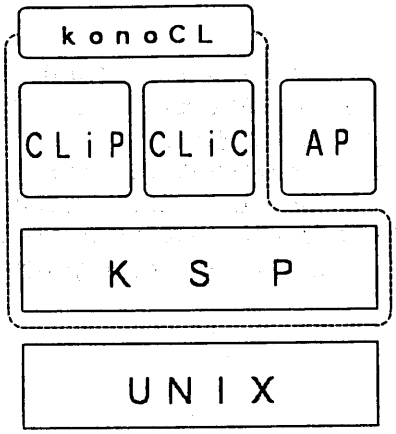


図１．　ｋｏｎｏＣＬの構成

kspは、CommonLispのみを意識したものではなく、記号処理を行うための核となるべき機能(メモリ管理、ローダ、システムサービス)を提供するものである。

CommonLispインタプリタCLiPは、kspのシステムサービスを利用しCommonLispのみで記述している。

CLiCも同様に、kspのシステムサービスと、CLiPが提供する関数の一部を用いて記述している。

CLiCは、CommonLispのcompile,compile-flle関数を提供するだけでなく、konoCL自身のシステムコンパイラであるため、リロケータブルでコンパクトなコードを出力することを目的としている。

本報告では、CLiCの構成、compile-codeの構成・実行方式、lambda-parameterの束縛方式、及び性能について述べる。

2. CLiCの構成

CLiCは、CommonLispのＳ式を含むファイルをコンパイルし、リロケータブルなオブジェクトファイル(faslファイル)を生成するファイル単位のコンパイラである。このfaslファイルは、kspのローダにより、CLiPの中に取り入れることができる。

CommonLispの、compile,compile-file関数は、CLiCの機能の一部を利用し実現している。

CLiCがソースファイルをfaslファイルに変換する過程を図２に示す。

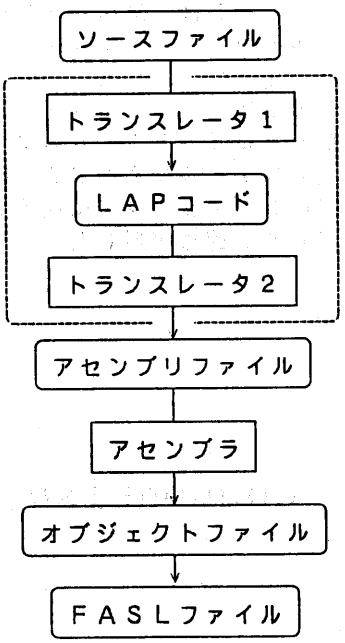


図２．　コンパイルの流れ

(1) Ｓ式を読み込み解析してCommonLispのストラクチャを用いた中間木に変換する。
(2) 中間木中の、ローカルな変数・関数のロケーションを決定する。
(3) 中間木をLAPコード(LAP命令とオペランドのリストの列)に変換する。
(4) トランスレータで、LAPコードを、68000系のアセンブリコードに変換する。
(5) UNIXのアセンブラでアセンブルし、UNIXのオブジェクトファイルを作る。
(6) オブジェクトファイルのヘッダを書き換えfaslファイルを作る。

3. faslファイルの構成

faslファイルは、ローダ用の制御命令と、関数本体の機械語コードで構成している。

ローダ制御命令は、バイトコードであり、ヒープの確保、機械語コードの登録、シンボルの登録や機械語コードの実行制御等を行う。

4. compile-codeの構成

kspは、faslファイルを読み込む際に、あらかじめ用意してあるbinary-code-areaにcomile-blockを確保し、その中に、それぞれの関数のcompile-codeの実体を設定する。

compile-code,compile-blockの構成を図3に示す。

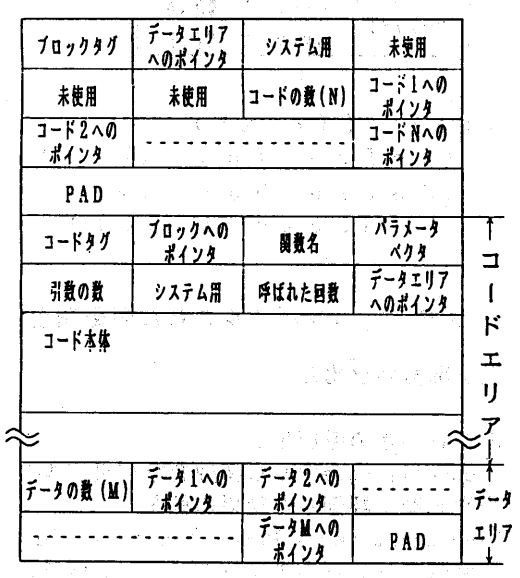


図３． compile-block の構成

compile-codeから参照するCommonLispのデータ(シンボルや文字列等)は、compile-code-blockごとにデータエリアが確保してあり、compile-code中からは、プログラムカウンタ相対で参照する。

5. compile-codeの実行方式

CLiCが生成するcompile-codeは、kspの環境の元で実行される。

5.1 スタック

kspは以下の３本のスタックで実行制御を行う。

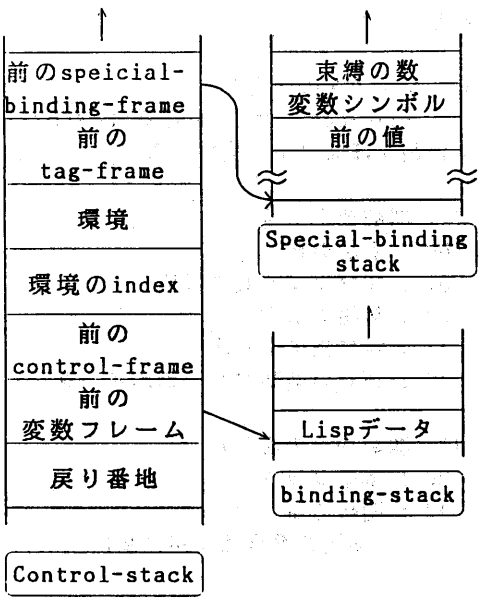


図４． スタックの構成

(1) control-stack

control-stackは、kspの核スタックであり、Ｃ言語のスタックと共存している。

compile-codeは、呼び出された際に、control-stack上にcontrol-frameを作成する。control-frameには、次の3種類がある。

- normal-frame ： 通常の呼出し
- closure-frame ： コード中で関数閉包を作る場合
- full-frame ： スペシャル変数の束縛、catch等の大域タグが存在する場合

(2) binding-stack

binding-stackは、関数の引数の受渡し、関数中のローカル変数の束縛やコード実行中のワークに使われる。このスタックには、CommonLispのデータ以外が積まれることはない。またcompile-codeは、コードの入口で、binding-stack上に変数フレームポインタを設定する。一つのコード中で、let

やflet等でローカルに変数や関数を束縛する場合にも、新しくフレームを設定することはない。

(3) special-binding-stack
スペシャル変数束縛時に、その変数シンボルと、束縛前の値を退避するために使用する。

5.2 レジスタの割当て
compile-codeでは、68000系CPUのレジスタ16個をほぼフルに使用する。

(1) 関数呼出し時
D0 : 引数の数
D1 : 呼び出す関数の環境
A0 : 最後の引数
A1 : 呼び出す関数名
A2 : 新しい変数フレームポインタ

(2) 関数からの戻り時
D0 : 多値フラグ
A0 : 関数の値
(多値の場合は最初の値)

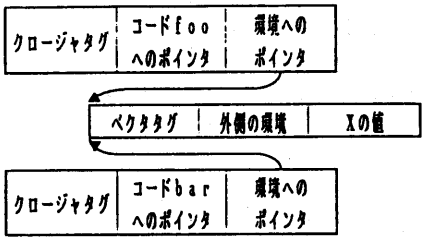
(3) コード実行時
D5 : control-frame のチェーン
D6 : ヒープ領域へのポインタ
D7 : NIL
A4 : kspのグローバルテーブル
A5 : ローカル変数フレーム
A6 : binding-stack pointer
A7 : control-stack pointer

5.4 関数閉包
kspのシンボルの構造を図5に示す。

値
インタフェース関数
eval用関数
関数定義本体
属性リスト
印字名
パッケージ
システム用

図5．シンボルの構造

CLiCは、CommonLispの関数やマクロをコンパイルすると、それぞれのシンボルの関数定義本体と、インタフェース関数の部分に、compiled-closure型のデータを設定する。
compiled-closureは、compile-codeと、そのコードが定義された時の環境で構成される。
環境は、CommonLispのsimple-vector型のデータで表現している。(図6)




```
(let ((x 1))
  (defun foo () x)
  (defun bar (y)
    (setq x y)))
```

図6． コンパイルされた関数閉包

5.3 関数呼び出し

5.3.1 通常の呼び出し
compile-code中から関数を呼び出す場合、引数を順に評価した結果の値をbinding-stack上に積む。その後、呼び出す関数名シンボルのインタフェース関数部分からクロージャを取りだし、呼び出し先の環境を設定してから、サブルーチンコールする。
呼び出す関数がcompile-codeの場合、インタフェース関数は、そのcomplie-codeからできた関数閉包になっているため、直接compile-codeを呼び出すことになる。
interpreter-codeや、kspが提供するexternal-codeを呼び出す際には、インタフェース関数が、必要なレジスタの退避、環境の設定などを行ったのち、実際の関数実行を行い、関数から戻る際には、元の環境を回復する。

5.3.2 局所関数呼び出し

通常の関数呼び出しでは、呼び出す関数が定義された環境を回復する必要があるので、関数閉包を経由して行う。

それに対し、fletやlabelsで定義される局所関数は、それが関数としてコードの外部へ出ていく(#'funの形式で参照される)ことのないかぎり、関数閉包を生成しないようにしている。そのため、呼び出し時には、局所関数が定義された環境を設定したのち、直接局所関数のコードへサブルーチンコールする。

5.3.3 末尾・再帰呼び出し

末尾再帰を除き、通常の再帰呼び出しは、上記した関数呼び出しとほぼ同じである。違いは、サブルーチンコールするアドレスが、コード中で静的に決定することができるので、関数シンボルからインタフェース関数を取り出す必要がないということである。

末尾呼び出しは、normal-frameを作るコードでのみ可能である。

末尾再帰は、上記条件のもとで、ループに変換する。

以上の関数呼び出しを実現するため、compile-codeの入口は3種類存在する。(図7)

```
        bra normal_point
        bra tail_point
normal_point:
        <control-frameの設定>
tali_point:
        [control-frameの拡張]
        <引数の数のチェック>
        <スタックオーバフローチェック>
trec_point:
        <キーボード割込みチェック>

        <コード本体>
```

図7. compile-codeの入口

5.5 変数

CommonLispの変数は、参照の方法により数種に分類できる。compile-code中でのそれぞれの変数の参照は以下のように行う。

(1)通常のローカル変数

lambda-parameterや、let等で確保されるローカル変数は、関数が呼び出された際に設定する変数フレームポインタからの相対で参照する。

(2)スペシャル変数(グローバル変数)

スペシャル変数は、lambda-parameterやlet等での束縛時に、大域シンボルの値に設定するので、束縛以後は、シンボルの値を直接参照することで実現できる。

(3)関数閉包で閉包される変数

関数閉包が生成される場合、閉包される変数を束縛する際に、binding-stack上に設定している環境の更新が行われる。それ以後の変数の参照は、環境であるsimple-vectorをインデックス参照することで行われる。

(4)特別なシンボル

nilは、レジスタに割当てられている。

tは、kspの持つグローバルテーブルから取りだす。

(5)変数フレームからの位置が静的に決定できない変数

①lambda-parameterの初期値で束縛されるローカル変数

```
#'(lambda (&optional x
                (y (let ((z (list x)))
                        (cons z z))))
          .....)
```

②multiple-value-callの第3引数以後に束縛されるローカル変数

```
(multiple-value-call #'foo
    (bar)
    (let ((x (baz))) (quux x x)))
```

③multiple-value-prog1の第２引数以後に束縛されるローカル変数

```
(multiple-value-prog1 (foo)
    (let ((x (bar))) (baz x x)))
```

これらの変数は、コード実行時により、binding-stack上での位置が異なる場合があり、コードの先頭で設定した変数フレームからの位置が静的に決定できない。

そこで、これらの変数は、その束縛時に新しくbinding-stack上に変数フレームを設定し直し、そのフレームからの相対で参照する。

5.6 多値

compile-codeが値を返す際には、必ず多値フラグを設定する。

多値を返す関数 values では、最初の値を値返却用のレジスタに設定すると共に、多値返却領域に全ての値を設定する。

こうすることで、通常の関数呼び出しの結果の値は、値返却用のレジスタを参照するだけでよい。

多値を扱う特殊形式のみが、多値フラグを参照して動作を決定する。

6. 関数閉包からの脱出

goとreturn-fromによる関数閉包からの脱出は、catch,throwによる大域脱出と全く同じ方式をとっている。

関数閉包からの脱出先のタグを持つ、blockやtagbody特殊形式では、そのコードの先頭で、脱出対象の各タグにユニークなラベル(gensym)を付け、現在の環境に追加した後、tag-frameを設定し、コードを順に実行する。関数閉包は拡張された環境を用いて生成するので、goとreturn-fromによる脱出は、環境から脱出先タグを取りだし、そこへthrowすることで実現している。

7. lambda-parameter の束縛

compile-codeでは、局所関数の呼び出しを除き、呼び出す先の関数のパラメータに関する情報がないため、引数を評価した結果をbinding-stack上に順に積んで、呼び出す関数に制御を渡す。

lambda-parameterの束縛は、呼び出された関数のcompile-codeのコード本体の先頭で行う。

CommonLispでは、optional parameter, rest parameter, keyword parameter, 及び supplied parameterの存在のため、渡って来た引数を、そのまま使用できないことがある。

また、スペシャル変数、関数閉包に取り込まれる変数は、binding-stack上から、globalシンボルの値や環境に移動する必要がある。

そのため、compile-codeの先頭では、

(1) 環境の設定
(2) 関数閉包に取り込まれる変数用領域の確保
(3) optional, keyword, supplied, aux parameter 用スタックの確保
(4) 引数の数によるoptional parameterの初期値の評価,supplied parameterの設定
(5) rest parameter用リストの生成
(6) :allow-other-keys のチェック
(7) 各keyword parameter の初期値及びsupplied parameterの設定
(8) aux parameterの初期値の設定

が行われる。

また、CLiCのcompile-codeでは、必要なパラメータだけをbinding-stack上に残し、全てのパラメータを変数フレームからの相対で参照できるように、それぞれのパラメータの値を設定する際に、binding-stack内で、渡ってきた引数を移動する。

以下の関数について、変数束縛部分のコードサイズ、実行時間を表１に示す。

```
(1) (defun foo ())
(2) (defun foo (x) x)
(3) (defun foo (&optioal x) x)
(4) (defun foo (&rest x) x)
(5) (defun foo (&key x) x)
(6) (defun foo (&aux x) x)
(7) (defun foo (&optioal (x 1 sp)) x)
(8) (defun foo (x)
          (declare (special x)) x)
(9) (defun foo (x) #'(lambda () x))
```

表１．変数束縛のメモリサイズと実行速度

関数	コードサイズ（バイト）	実行時間（秒/100000回）
1	22	0.06
2	22	0.06
3	60	0.32
4	30	0.18
5	172	1.44
6	32	0.16
7	102	0.62
8	114	1.60
9	198	1.84

HP-9000/350(MC68020,25MHz,4.0MIPS)で測定

コードサイズには、引数の数のチェック及びエラーハンドラへの分岐のコードが含まれている。

rest parameter用のリストの作成、keyword parameter用のチェック、スタックの走査部分はサブルーチン化している。

スペシャル変数の束縛(8)、関数閉包作成のための環境の更新(9)をインラインでコード中に出力しているためコードサイズが増大している。

8. compile-codeの性能

CLiCでは、リロケータブルなコードを出力し、compile-codeからの関数呼び出しは全てデータエリア中のシンボルを経由する。

また、レジスタを専用に割り当てているので、kspが提供するシステムサービスを利用するためには、実行環境の切り替えを必要とする。そのため、Lisp中で閉じた関数の速度に比べ、システムサービスを呼び出す処理(I/O等)は多少遅くなる。

表２に、gabrielのベンチマーク[3]の測定結果を示す。

表２．gabriel benchmark 測定結果

#	ベンチマーク	実行時間(秒)
1	tak	0.40
2	stak	2.76
3	ctak	1.78
4	ltak	2.34
5	Boyer	7.96
6	Browse	35.16
7	destructive	6.10
8	traverse	38.80
9	derivative	2.24
10	D-deriv	2.50
11	Div-I	0.90
12	Div-R	0.88
13	FFT	17.82
14	fprint	2.50
15	fread	3.40
16	tprint	3.86

HP-9000/350で測定

9. 今後の課題

現在のCLiCは、核になる関数のインライン、宣言を利用した型チェックの除去程度の最適化しか行っていない。

今後

(1) 述語関数利用による型伝搬

(2) 局所関数の展開

(3) lambda-parameterの解析の強化

を行っていく予定である。

参考文献

1) 中村他："konoCL -kono Common Lisp-"，情報処理学会 記号処理研究会資料42-3，1987

2) 松島："多重領域をサポートした意味解析言語"，情報処理学会自然言語処理研究会資料61-3，1987

3) R.P.Gabriel："Performance and Evaluation of Lisp Systems"，MIT Press，1985

＊：ＵＮＩＸオペレーティングシステムは、米国ＡＴ＆Ｔ社が開発したソフトウェアであり、ＡＴ＆Ｔ社がライセンスしています。

付録

```
(defun foo (x &optional (y 1 sp) z &rest l &aux (u (list x z l)))
  (declare (special sp z))
  (list #'(lambda () y) sp u) )
```

```
(LAP G11 0x00020004)                    special変数が２個、普通の変数が４個の
                                        関数の始まりの宣言
(SETFULLCFRAME 3)
(ANUMSAVE 3)
(REQPARAM 1)                            引数は１個以上か？
(SETSVFRAME 0)                          special binding 用のフレームの準備
(RESTCOPY 3 4)                          binding stackの３個目の引数以後を、
                                          Fpからのオフセットが４以後へコピーする
(ENVMOVE 3)                             closureの環境をFpからのオフセットが３の
                                          ところにセットする
(UPDATELOCALENV 1 3)                    １個分の場所をとり、環境を更新する
                [ optional parameter の処理開始 ]
(TST (R R_ANUMBER))                     y に対応する引数は渡ってきたか
(JMPZ G14)                              なければ、ラベルへブランチ
(MOVE (S 1) (H 0 1 3))                  y の値を環境へ移動
(SPECMOVE (G _T_HALFCELL) (L 8))        spをＴに束縛し、スペシャル変数とする
(SUB 1 (R R_ANUMBER))
(TST (R R_ANUMBER))                     z に対応する引数は渡ってきたか
(JMPZ G13)                              なければ、ラベルへブランチ
(SPECMOVE (S 2) (L 9))                  z をスペシャル変数にする
(SUB 1 (R R_ANUMBER))
(JMP G15)                               [ optional parameterは全て渡ってきた ]
(LAB G14)
(MOVE (I 0) (H 0 1 3))                  y の初期値を１に設定
(SPECMOVE (R R_NIL) (L 8))              spをNILに束縛し、スペシャル変数とする
(LAB G13)
(SPECMOVE (R R_NIL) (L 9))              z をNILに束縛し、スペシャル変数とする
(MOVE 0 (R R_ANUMBER))
(LAB G15)
                [ optional parameter の処理終了 ]
(ANUMRESTORE)
                [ rest parameter の処理開始 ]
(RESTLIST 1)                                残った引数をリストにして１を束縛する
                [ rest parameter の処理終了 ]
                [ aux parameter の処理開始 ]
(PUSH (S 0))                            x をスタックに積む
(SYMBOLVALUE (L 9) (R R_RETVAL))
(PUSH (R R_RETVAL))                     スペシャル変数 z の値をスタックに積む
(PUSH (S 1))                            １をスタックに積む
(LIST 3 (R R_RETVAL))
(MOVE (R R_RETVAL) (S 3))               スタックトップの３個でリストを作り、u を
                                          束縛する
                [ aux parameter の処理終了 ]
.............
```