ISO標準Lisp言語ISLISPのインタプリタおよびコンパイラ

泉 信人† 伊藤貴康†

ISLISPはLisp言語のISO標準言語である．ISLISPのインタプリタとコンパイラを試作し，TISLと名付けた．TISLシステムとそのベンチマークプログラムによる評価結果について報告する．ISLISPはScheme並みにコンパクトなCommon Lisp系の言語でオブジェクト指向機能を備えている．TISLインタプリタは入力されたプログラムを評価形式ごとに一度中間コードに変換しながら解釈実行を行う．TISLコンパイラは中間コードをC言語のプログラムに変換し，インタプリタよりも高速に動作する1つの実行ファイルを作成するために使用される．また，TISL処理系全体がC言語で記述されており，パソコンやワークステーションへの移植性にも優れている．ISLISPインタプリタとしてはOpenLispが存在するが，TISLインタプリタはOpenLispよりも1.3~3.3倍高速であり，TISLコンパイラはTISLインタプリタよりも1.0~5.5倍高速である．

Interpreter and Compiler of the ISO Standard Lisp ISLISP

NOBUTO IZUMI† and TAKAYASU ITO†

ISLISP is the ISO standard Lisp language. We implemented its interpreter and compiler, called the TISL system. In this paper, after explaining an outline of the TISL system, we report its experimental results, using Gabriel benchmark programs. The TISL system is implemented in the C language so as to allow TISL portable for various PCs and workstations. The TISL interpreter first transforms a form into intermediate codes, and the resultant intermediate codes will be actually interpreted and executed. The TISL compiler is realized as a compiling function that can be invoked under the interpreter and it compiles intermediate codes into C programs. For efficient implementations of object-oriented features of ISLisp we introduce "type inference" in implementing generic functions. Compared to OpenLisp (an ISLISP interpreter), the TISL interpreter is 1.3 ~ 3.3 times faster than OpenLisp, and the TISL compiler is 1.0 ~ 5.5 times faster than the TISL interpreter.

1. はじめに

プログラミング言語LISPは，人工知能や記号処理の研究のために1958年に誕生した言語で，FORTRANに劣らない長い歴史を持っている．最初の実用的なLISP言語であるLISP1.5以来，諸種の言語機能の拡張が行われ，様々なLISP方言が生まれた．LISPはエキスパートシステムや数式処理システムの実現に用いられ成功を収めているが，1980年代になって，産業用言語としての国際標準を制定する気運が高まった．国際標準化の活動[3]が1987年以来行われ，1997年5月にLisp言語のISO標準としてISLISPが制定され[1]，1998年7月にJIS標準が制定された[2]．

ISLISPは多重名前空間を持つCommon Lisp系のLisp言語でありながら，Scheme並みにコンパクトにまとめられ，Common Lisp[7]のCLOSを参考に設計されたコンパクトなオブジェクト指向機能を備えている．ISLISPの最初の処理系としては，OpenLispが発表されているが[4]，機能上のバグがあり，また，OpenLispはインタプリタであるためコンパイラの実装が期待されていた．このことを念頭に，筆者らはISLISPのインタプリタとコンパイラを作成し，この処理系をTISL（Tohoku university ISLisp）と名付けている．ベンチマークの結果によれば，TISLインタプリタは機能と性能の両面でOpenLispより改善され，性能的にはOpenLispより1.3~3.3倍高速である．また，TISLコンパイラは，TISLインタプリタよりも1.0~5.5倍高速である．なお，TISLコンパイラは稼動しているISLISPコンパイラとしては，筆者らの知る限りでは，最初のシステムである．

本稿では，ISLISPの概要について説明した後，TISL処理系の概要について説明し，処理系の実験評価の結

† 東北大学大学院情報科学研究科
Department of Computer and Mathematical Sciences, Graduate School of Information Sciences, Tohoku University

果を報告する．

2. ISLISP

Lisp 言語の ISO 標準として制定された ISLISP は日本が提案した核言語を基に設計されたものである[3]．ISLISP は Common Lisp 系の Lisp 言語でありながら，Scheme 並みにコンパクトにまとめられた言語となっており，Common Lisp[7]の CLOS を参考に設計されたコンパクトなオブジェクト指向プログラミング機能を備えている．ISLISP☆は 6 つの名前空間（変数，動的変数，関数，クラス，ブロック，タグ）を持つ多重名前空間方式を採用している．変数束縛は基本的に静的束縛であるが，(dynamic *var*) と明記することにより変数 *var* を動的変数として扱うことができる．

2.1 評価モデル

ISLISP の評価には，2 つの段階がある．ISLISP テキストは，まず実行準備され，次にその準備されたテキストが実行される．実行準備されたテキストは，次のように実行される．

A. 評価形式がリテラルの場合，評価形式そのものを結果とする．

B. 評価形式が識別子の場合，現在の静的環境での変数名前空間でその識別子が指定するオブジェクトを結果とする．

C. 評価形式が複合形式の場合，次のように扱われる．

(1) 演算子が特殊演算子の場合，引数は特殊演算子の定義に従って評価される．

(2) 演算子が定義演算子の場合，第 1 引数が識別子であり，残りの引数が定義演算子の定義に従って扱われ，その結果のオブジェクトが適切な名前空間において識別子に束縛される．

(3) 演算子がラムダ式の場合，引数が左から右の順で評価される．その後，評価した結果を実引数としてラムダ式で指定された関数を呼び出す．

(4) その他の場合，複合形式は関数適用形式となる．評価形式の演算子の位置の識別子が現在の静的環境の関数名前空間で評価され，呼び出すべき関数を生成する．引数が左から右の順に評価され，その値を実引数として関数を呼び出す．

2.2 オブジェクト指向

ISLISP のオブジェクト指向機能は Common Lisp の CLOS を参考にしながらもコンパクトに設計されており，CLOS と同様にクラスと包括関数（generic function）に基づくオブジェクト指向機能がある．なお，CLOS から削除されたものとしてクラス変数や局所的な包括関数などがあげられる．

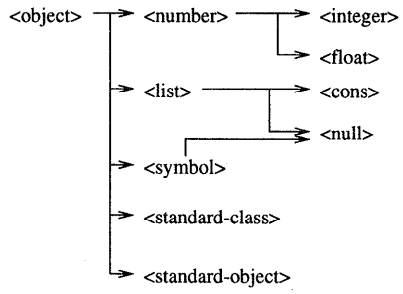


図 1 ISLISP におけるクラス継承グラフの例
Fig. 1 An example of inheritance graph in ISLISP.

2.2.1 クラス

ISLISP のデータ型はクラスシステムとして実現されており，すべての ISLISP オブジェクトはあるクラスのインスタンスである．クラスは <object> を根とする継承グラフを構成し，制限されているが多重継承も可能となっている．メタクラスとして <built-in-class> と <standard-class> が用意されている．**図 1** は ISLISP の定義済みクラスの継承グラフを一部図示したものである．<object> を上位クラスとして各クラスが存在している．<null> は <symbol> と <list> を多重継承している．<standard-object> は <standard-class> のインスタンスであり，<standard-class> のインスタンスである他のすべてのクラスの上位クラスとなる．

ユーザがクラスを定義するときには defclass 定義形式を次のように使用する．

```
(defclass <user-class> (<standard-class>)
          ((slot :reader get-slot)) (:abstractp nil))
```

クラスは，クラス名 (<user-class>)，上位クラスのリスト ((<standard-class>))，スロット指定 (((slot :reader get-slot)))，クラス任意機能 ((:abstractp nil)) からなる．クラスには複数の名前付きのスロットを定義することが可能であり，そのクラスのインスタンスはスロットに値を 1 つ保持させることが可能である．スロットへのアクセス関数はクラスの定義時に指定する．また，クラスの定義時に上位クラスを指定しスロットの構造を継承することができる．この上位クラスからクラス優先度リストを作成する．

クラス優先度リストとは自分自身を含めた上位クラスに関する全順序であり，上位クラスの局所優先順序

☆ 以下の ISLISP についての説明は文献 1), 2) に基づく．用語を含め，詳細はこれらの文献を参照されたい．

と矛盾しないものとなっている．すなわち，あるクラスのクラス優先度リストとは自分自身を先頭とし，上位クラスのクラス優先度リストを順に連結したものとなる．このとき，重複するものは左側のものを削除する．ここで，ISLISP の標準クラスにおける多重継承の制限により <object > と <standard-object> 以外のクラスが重複することは違反となる．クラス優先度リストは包括関数の呼び出しなどの際に使用される．

2.2.2 包括関数

包括関数とは引数のクラスによって呼び出し時の振舞いが異なる関数である．包括関数は引数が特殊化されたメソッドによって構成されており，包括関数が呼び出されると，実引数に従って適用可能なメソッドが選択される．このとき，実引数のクラスのクラス優先度リストに従って並べられた実効メソッドを決定する．そして包括関数の実引数がそのまま渡されて実効メソッドが起動され，包括関数の値を計算する．ここで適用可能なメソッドとは，すべての引数に対して実引数のクラスがメソッドの仮引数に指定されたクラスの下位クラスとなっているメソッドのことをいう．また，次の優先度のメソッドを呼び出す場合にはメソッドの中で call-next-method 局所関数を呼び出す．

ユーザは包括関数およびメソッドを，それぞれ，defgeneric および defmethod を用いて定義できる．オブジェクトの生成には包括関数 create を使用する．

3. ISLISP 処理系 TISL の概要

TISL インタプリタは入力されたソースプログラムを直接解釈実行するのではなく，評価形式ごとに，一度中間コードに変換してから解釈実行する．TISL コンパイラはソースプログラムを中間コードに変換してから，この中間コードを C 言語のプログラムにコンパイルし，その後，結果を C 言語のコンパイラによってコンパイルする．また，TISL 処理系全体は C 言語によって記述されており，PC や UNIX ワークステーションでも実行可能な移植性に優れたシステムとなっている．**図 2** に TISL 処理系の構成図を示す．

TISL インタプリタではユーザから入力されたテキストはリーダによって評価形式ごとに内部表現に変換される．その内部表現は変換部によって中間コード形式の関数オブジェクトに変換される．関数オブジェクトは実行・制御部によって実行され，実行結果がライタを通してユーザに表示される．この間に作成されるオブジェクトはメモリ管理部によって管理されるヒープ領域に作成される．ヒープ領域は固定長のセルからなる固定長領域と任意の大きさの領域を割り当てられる不定長領域からなり，それぞれ別々の GC 戦略を用いている．

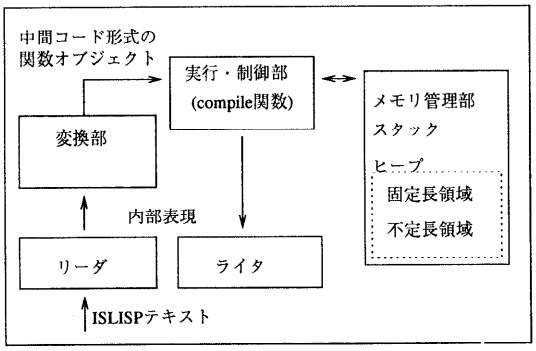


図 2 TISL 処理系の概要
Fig. 2 Outline of TISL system.

TISL コンパイラは，TISL インタプリタの実行・制御部に用意されているコンパイル関数 compile によって実現されている．コンパイルの対象となる評価形式は，初めに，インタプリタの変換器によって中間コード形式の関数オブジェクトに変換される．その後，compile 関数は関数オブジェクトを C 言語プログラムへとコンパイルを行う．得られた C 言語プログラムとすでに作成されている共通プログラムを一緒に C コンパイラによってコンパイルし，アプリケーションとして利用可能な独立した実行ファイルを作成することが可能である．これが実行されると，コンパイル時に抜き出されたオブジェクトの環境初期化用の関数を実行し，環境の初期化を行う．その後，対応する C 言語の関数を呼び出し結果の値を表示する．リーダ，ライタ，メモリ管理部などはインタプリタと共通のものを使用している．

4. TISL インタプリタ

TISL インタプリタは入力テキストを一度中間コードに変換してから解釈実行する方式を採用している．リーダが最上位の評価形式を 1 つ読み込むたびに内部表現に変換し，結果を中間コード形式の関数オブジェクトに変換する．その関数オブジェクトを実行し，結果の値を出力する．関数オブジェクトへの変換は次のようなステップで行われる．

(1) テキストから内部表現を作成
(2) 内部表現から中間コードリストへの変換
(3) 中間コードリストから関数オブジェクトを作成

関数オブジェクトの作成とその実行においては局所変数の扱い，最適化，ガーベジコレクション（GC）に加えて，ISLISP のオブジェクト指向機能を実現する

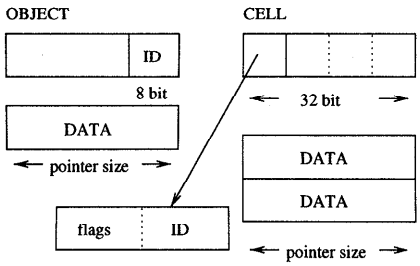


図 3 OBJECT と CELL の構造
Fig. 3 OBJECT and CELL.

核となる包括関数の実現が問題となる．この章では，ISLISP テキストの内部表現，中間コード，関数オブジェクトとその実行，GC の実現法，および，インタプリタにおける包括関数の実現法について説明する．

4.1 TISL における内部データ構造

ISLISP テキストを処理系内部で表現するために次のデータ構造単位を使用する（図 3 参照）．**DATA** は整数，浮動小数点数，ポインタなどを保持できる C 言語の共有体であり，サイズはポインタのサイズとしている．この共有体にはタグが付けられていないため DATA だけではこれが何を表現しているかを知ることはできない．そこで，タグを付け DATA が何を表現しているか区別する **OBJECT** 構造体を用意する．演算を行うスタックは OBJECT 構造体の配列によって表現されている．このほかコンスを作成するために 1 つのタグと 2 つの DATA からなる **CELL** 構造体がある．CELL はメモリ管理部で管理される固定長領域のデータの単位として使用される．CELL はタグの一部を用いて，その CELL が何を表しているのかを示す ID と GC などに用いるフラグを保持している．これらの構造体を用いることにより，32 bit あるいは 64 bit などのマシンアーキテクチャの違いも DATA 共有体のサイズを変更するだけでよく，移植が容易なものとなっている．また，浮動小数点数に対して特別な表現をすることなく C 言語の float あるいは double を利用して表現されており，浮動小数点数を用いた演算を比較的高速に演算することが可能となっている．

これらのデータ構造単位を使用して Lisp オブジェクトを構成する．<integer>，<float> のインスタンスは OBJECT 構造体のみによって表現される．CELL などを使用するオブジェクトは OBJECT に保持されたポインタを通して参照される．図 4 に CELL を使用するオブジェクトを示す．<cons> のインスタンスは CELL 構造体を用いて表現する．CELL の 2 つの DATA を用いて car，cdr 部の値を保持させ，それぞれの DATA の型を CELL のタグに保持させる．内部

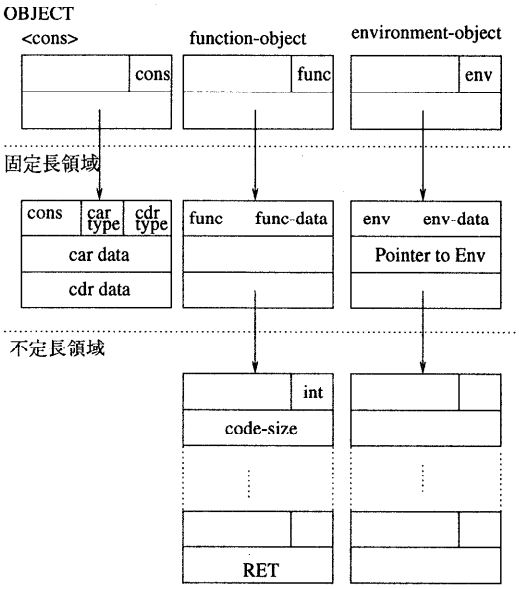


図 4 データ表現
Fig. 4 Data presentations.

の実行単位となる関数オブジェクトも CELL を使用して作成する．CELL のタグに関数の引数の数などを保持させ，不定長領域から確保した中間コード配列へのポインタを DATA に保持する．また，実引数を保持するために作成される環境オブジェクトも CELL を使用して作成する．CELL のタグに環境オブジェクトが保持する値の数を記録し，この環境オブジェクトが作成されたときの環境オブジェクトへのポインタを 1 つの DATA で参照する．実引数を格納するために不定長領域から確保した OBJECT 配列へのポインタをもう 1 つの DATA に保持する．このほかに GC の対象となる Lisp オブジェクトは CELL と不定長領域を用いて表現している．GC の対象とならないオブジェクトは専用の構造体を用いて表現している．

ISLISP では <standard-object> 以外の定義済みクラスの扱いは実装に依存する組込みクラスとしてもよいと定義されている．TISL では<standard-object> 以外の定義済みクラスはすべて組込みクラスとして実装しており，これらのクラスの下位クラスをユーザが作成することはできない．また，ISLISP では定義形式は同じ名前に対して 2 回以上現れてはならないとなっているので，TISL ではクラスや包括関数の再定義を許していない．よって，クラスや包括関数は一度定義されると GC の対象外のオブジェクトとなる．標準クラスは専用の構造体を用意して作成され，この構造体はクラスの名前，クラス優先度リスト，スロット記述，クラス修飾子からなる．インスタンスは CELL

を使用し，自分のクラスへのポインタと不定長領域から確保したスロット用の値を保持する配列へのポインタによって表現される．包括関数を表す構造体は引数の数と自分を構成するメソッドのリストと実効メソッドを保持するための表からなる．メソッドは引数を特殊化しているクラスの配列と本体部から作成した関数オブジェクトからなる．実効メソッドはメソッドを修飾子別に優先度順に並べたものとなっている．

4.2 中間コード

TISLで用いている中間コードは基本的に後置コードであり，スタックを用いて演算を行う．各命令は，リテラルオブジェクトをスタックに積む命令，ISLISPの定義済みの関数に対応しスタック上の値を用いて演算を行う命令，ユーザ定義の関数を呼び出す命令，条件分岐などを行う制御用の命令などに分類できる．**表1**にそれらの命令を示す．TISLでは表の命令を含めて300種以上の命令がある．中間コードへの変換はISLISPの評価モデルの評価順に従って行われるが，関数適用形式の演算子は静的に決定できるため，組込み演算子の場合には対応する命令に変換する．

4.3 関数オブジェクトの作成と実行

この節では，テキストから中間コードリストへの変換について説明した後，関数オブジェクトの作成の仕方について説明する．**図5**はfact関数のプログラムテキストと中間コードリストへの変換の例である．図5中のtop-level-function：body-listは最上位の評価形式を変換して作成された中間コードリストを示している．この中間コードリストは1つのリストとして作成される．fact：function-list以下のリストは{ fact：function-list }の部分に位置している．変換過程をこのfact関数を例にとり説明する．最後に関数オブジェクトの実行について説明する．

4.3.1 中間コードリストへの変換

TISLインタプリタは最上位の評価形式ごとに解釈実行する．最初にリーダがユーザの入力した最上位の評価形式のテキストをTISLの内部表現によるリストに変換する．このリストから中間コードのリストをISLISPの評価モデルに従って次のように作成する．

A. 評価形式がリテラルである場合
リテラルオブジェクトをスタックに積む命令に変換する（例，**PUSH-INTEGER** 1）．

B. 評価形式が識別子である場合
変換器は局所的な名前解決用のリストを持っており，識別子の静的な環境から次のように処理する．
(1) 大域変数の場合
大域の変数名前空間の値をスタックに積む命令に変換する．
(2) 局所変数の場合
局所変数は，この時点ではスタック上で扱われるかヒープ上で扱われるかを決定できないので，仮の命令（**PUSH-LOCAL-VARIABLE** *symbol*）に変換しておく．この局所変数を名前解決用リストに追加した仮引数リストが，この時点での最も局所的なものでない場合には，この局所変数を追加した仮引数リストにHEAPのマークを付ける．

表1 実行部で実行される命令の例
Table 1 Examples of intermediate commands.

命令	動作
PUSH-INTEGER *i*	整数 i をスタックに積む．
PUSH-CONS *p*	コンス p をスタックに積む．
PUSH-STACK *offset*	スタックトップから *offset* の位置の値をスタックに積む．
PUSH-HEAP *offset*	環境オブジェクト上の値をスタックに積む．
CONS	スタックトップの要素をcdrとしその次の要素をcarとするコンスを作成しスタックに積む．
NUMBERLESS	スタックトップの値 st とその次の要素 nv を取り出し，$nv < st$ ならt，$nv \geq st$ ならnilをスタックに積む．
ADDITION	スタックトップとその次の要素を取り出し，和の値を計算しスタックに積む．
SUBTRACTION	スタックトップの値 st とその次の要素 nv を取り出し，差 $nv - st$ を計算しスタックに積む．
MULTIPLICATION	スタックトップとその次の要素を取り出し，積の値を計算しスタックに積む．
DEFUN *name func*	引数に識別子と関数オブジェクトをとり，大域の関数名前空間の束縛を作成する．
CALL *symbol n*	*symbol* と名付けられた大域関数を引数 n 個で呼び出す．
CALL-GLOBAL-STACK *func*	引数をスタック上で扱える大域関数 *func* を呼び出す．
IF *then-form else-form*	*then-form*, *else-form*を変換した関数オブジェクトを引数にとり，スタックトップの値で分岐を行う．
RET	関数オブジェクトの実行を正常終了する．
NUMBERLESS-STACK-INTEGER *offset i*	スタックトップから *offset* の位置の値 v と整数 i の大小比較を行い．$v < i$ ならt，$v \geq i$ ならnilをスタックに積む．
SUBTRACTION-STACK-INTEGER *offset i*	スタックトップから *offset* の位置の値 v と整数 i の差 $v - i$ の値を計算しスタックに積む．

```
(defun fact (n)
  (if (< n 2)
      1
      (* n (fact (- n 1)))))
```

(a) 階乗を計算する関数 fact

```
top-level-function : body-list
(
 DEFUN fact { fact : function-list }
 RET
)

fact : function-list
(;parameter-list
 ((1 . STACK) n)
 ; body-list
 PUSH-LOCAL-VARIABLE n
 PUSH-INTEGER 2
 NUMBERLESS
 IF { then-form : body-list } { else-form : body-list }
 RET
)

then-form : body-list
(
 PUSH-INTEGER 1
 RET
)

else-form : body-list
(
 PUSH-LOCAL-VARIABLE n
 PUSH-LOCAL-VARIABLE n
 PUSH-INTEGER 1
 SUBTRACTION
 CALL2 fact
 MULTIPLICATION
 RET
)
```

(b) fact に対する中間コードリスト

図5 中間コードの作成の例

Fig. 5 An example of constructing intermediate codes.

C. 評価形式が複合形式である場合

演算子の位置にラムダ式がある場合には，評価形式の引数を左から右の順に変換していき，最後にラムダ式を変換して作成した関数オブジェクトを呼び出す仮の命令（**CALL-LAMBDA** *function*）に変換する．演算子の位置がシンボルである場合には呼び出すべき関数は静的に決定できるので関数の種類によって次のように変換する．

(1) マクロの場合

マクロ展開をした評価形式を変換する．

(2) 局所関数の場合

引数を左から右の順に変換していき，最後に局所関数を呼び出す仮の命令（**CALL-LOCAL** *symbol n*）に変換する．

(3) 特殊・定義関数の場合

演算子が特殊演算子，定義演算子の場合にはそれぞれの定義に従って変換を行う．たとえば，defun 定義演算子は引数の識別子と大域関数の束縛を作成する命令であり，引数の評価は行われない．よって，引数から作成した関数オブジェクトのためのリスト（引数のリストと中間コードリスト）を命令 **DEFUN** の引数として用い，命令（**DEFUN** *symbol function*）を作成する．

(4) 大域の関数適用形式の場合

引数を左から右の順番に変換していき，その後に，関数名前空間で *symbol* と名付けられた関数を呼び出す命令（**CALL** *symbol n*）に変換する．

また，メソッドは call-next-method 局所関数を用いて次の優先度のメソッドを呼び出すことができる．このため，メソッドの中間コードリストを作成する段階で call-next-method 局所関数を参照するか否かを検査しておく．

図 5(a) の評価形式を以上のように変換すると，図5(b)のような中間コードリストが作成される．最上位の評価形式の演算子 defun は定義演算子であり，引数から識別子と関数オブジェクトのためのリストが作成される．関数オブジェクトのためのリストは引数のリストと中間コードリストからなり，引数のリストは引数の数とこれらの引数がスタック上で扱えるかどうかを検査するためのマークと引数名のリストからなる．この関数オブジェクトのためのリストは後で関数 fact を呼び出したときに実行される関数オブジェクトに変換される．関数 fact の最初の評価形式（if *test-form then-form else-form*）の演算子 if は特殊演算子であり，最初に *test-form* から中間コードリストを作成し，その後に条件分岐を行う命令 **IF** を作成する．この命令は引数に *then-form* と *else-form* 用の関数オブジェクトをとるが，この段階では中間コードリストを作成して引数の代わりに用い，後で関数オブジェクトに変換する．各中間コードリストの最後には中間コードの終わりを表し，関数オブジェクトの実行を正常に終了させる命令 **RET** が追加される．

4.3.2 局所変数

局所的な変数は，その値がさらに局所的な関数によって参照される場合には，その関数の存在期間が終了するまで，値にアクセスできるようにしておく必要がある．この場合，ヒープ上に環境オブジェクトを作成し，スタック上の値をこの環境オブジェクトにコピーして操作を行う．これ以外の場合には，変数のスコープの消失とともに値を残しておく必要はないのでスタック

上でそのまま扱い，環境オブジェクトの作成のコストや参照時に環境オブジェクトのリンクをたどるコストが削除できる．また，環境オブジェクト用にセルを消費しないのでGCの発生も抑制され，効率的な実行が可能となっている．

4.3.3 関数オブジェクト

関数オブジェクトはTISL処理系で実行の単位であり，引数の数，引数がスタック上で扱えるか否かの情報，および，中間コードから構成されている．4.3.1項に述べた変換によって作成された中間コードリストから関数オブジェクトを作成する．

最上位の評価形式は引数なしの関数オブジェクトとして作成される．TISLインタプリタでは各命令を実行するC言語の関数が用意されており，関数オブジェクトの中間コード配列にはこの関数へのポインタを保持させて，実行時にこの関数を順番に呼び出すことによって中間コードの実行を行う．中間コードリストから1つずつ命令を抜き出し関数オブジェクトの中間コードを作成していく．このとき，実行時のスタックの動きと環境オブジェクトの動きをトレースしておき，局所変数を参照する命令 **PUSH-LOCAL-VARIABLE** を適した命令（スタックの場合：**PUSH-STACK**，ヒープの場合：**PUSH-HEAP**）に変換し，引数の *offset* を計算する．この他，関数オブジェクトの引数をスタック上で扱うかヒープ上で扱うかが決定していなかった命令を決定する（**CALL-LAMBDA** → **CALL-LAMBDA-STACK** または **CALL-LAMBDA-HEAP**）．

メソッドの関数オブジェクトを作成するとき，call-next-method局所関数を参照している場合には局所関数を実行時に用意する必要があるので，**MAKE-NEXT-METHOD** という局所関数を作成する命令を先頭に埋め込む．call-next-method局所関数を参照していない場合にはこの命令は必要なく，局所関数を作成するためのコストが削除できる．

図6 は4.3.4項で述べる最適化を **NUMBERLESS** と **SUBTRACTION** に対して行って，図5から得られた関数オブジェクトの例である．図6中のtop-level-function，factなどは図5から得られる関数オブジェクトを表現しており，Parameter-Numberは仮引数の数，Parameter-Stackは実引数がスタックで扱えることを意味している．

4.3.4 最適化

TISLインタプリタではユーザと対話しながらプログラムの実行を行っていくため時間のかかる最適化処理は行わずに，**図7** のようなスタック操作に関する単純な最適化を行うにとどめている．

```
top-level-function : Parameter-Number 0
DEFUN fact { fact : function }
RET

fact : Parameter-Number 1, Parameter-Stack
NUMBERLESS-STACK-INTEGER 0 2
IF { then-form : function } { else-form : function }
RET

then-form : Parameter-Number 0
PUSH-INTEGER 1
RET

else-form : Parameter-Number 0
PUSH-STACK -1
SUBTRACTION-STACK-INTEGER -1 1
CALL-GLOBAL-STACK { fact : function }
MULTIPLICATION
RET
```

図6 中間コード形式の関数オブジェクトの例
Fig. 6 An example of functional objects in the intermediate codes.

```
PUSH-STACK offset
PUSH-INTEGER 2
NUMBERLESS
      ⇓
NUMBERLESS-STACK-INTEGER offset 2
```

図7 最適化の例
Fig. 7 An example of code optimization.

図7のようにスタック上の値とリテラル定数との演算を行う命令系列の場合，スタック上に値を積まずに直接演算を行うと3ステップの命令が1ステップになり，さらに，スタックに値を積まずに演算を行うので実行コストが軽くなる．すべての命令に対してこのような命令を用意すると膨大な数になってしまうので，現在では述語命令や算術演算命令に対して **NUMBERLESS-STACK-INTEGER** や **SUBTRACTION-STACK-INTEGER** のような命令を用意している．スタック上の値とリテラル定数の演算の場合に，図7のようにスタック上に値を積まない命令を生成する最適化を行っている．図6はこのような最適化を行った例である．

defunで作成した関数の中で自分自身を呼び出す場合，**CALL** を使用した識別子経由による関数呼び出しではなく，直接，関数オブジェクトを指定して

関数を呼び出す命令を出力した方が実行コストが小さい．図5(b)の **CALL2** は fact 関数の再帰呼び出しであるが，次の中間コードに変換する段階で関数オブジェクトが決定できるのでこの関数オブジェクトを使用して **CALL-GLOABL-STACK**（または，**CALL-GLOBAL-HEAP**）に変換する．図6の例では，図5(b)に対しこれらの最適化も行っている．

4.3.5 関数オブジェクトの実行

最上位の評価形式を変換して作成した関数オブジェクトを除いて，関数オブジェクトは **CALL** などの命令から呼び出される．関数オブジェクトが呼び出されるときには実引数がスタックに積まれており，関数オブジェクトの引数がスタック上で扱えない場合には環境オブジェクトを作成し，スタック上の値をコピーする．スタック上で扱える場合にはこのまま処理を続ける．

次に関数オブジェクトの中間コードを実行していく．関数オブジェクトの中間コード配列には各命令に対応する C 言語の関数へのポインタが保持されており，この関数を順に呼び出すことによって関数オブジェクトの実行を行う．各 C 言語の関数は返り値によって命令の実行が正常終了したか異常終了したかが判別でき，関数オブジェクトの中間コード配列の実行は命令が正常終了するあいだ続けられ，異常終了すると原因が調べられる．

RET は自分を異常終了させて関数オブジェクトの実行終了を伝える．このときスタックトップに関数オブジェクトの実行結果の値が積まれており，引数の数だけスタックを下げ，関数オブジェクトの実行結果の値をスタックトップに積み直す．その後，この関数オブジェクトを呼び出した命令は正常終了する．

非局所脱出の場合，脱出を行う命令は自分を異常終了させて非局所脱出を知らせる．関数オブジェクトを呼び出す命令は非局所関数を扱えないので自分を異常終了させ，自分を呼び出している命令に異常終了を伝える．非局所脱出の脱出先を扱う命令は異常終了の原因から自分への脱出か否かを調べて対応する処理を行う．

命令の実行で何らかの例外が発生するとき，その場で例外を引数としてハンドラ関数が呼び出される．この関数は必ず異常終了し，適した非局所脱出が行われる．初期のハンドラ関数は，ISLISP では処理系定義となっており，TISL 処理系では例外の表示を行って最上位まで非局所脱出を行う．

最初の関数オブジェクトを実行し，**RET** で終了した場合にはスタックに結果の値が1つ積まれた状態になるので，この値をユーザに表示する．

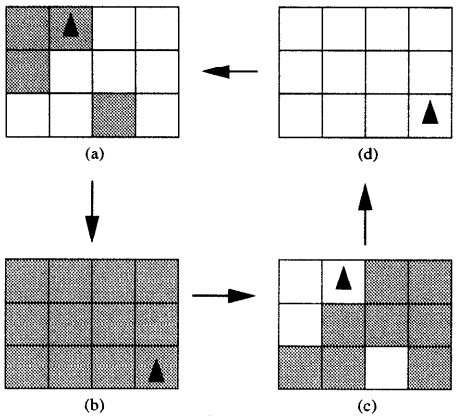


図8 固定長領域の GC
Fig. 8 GC for cells with fixed length.

4.4 GC の実装法

LISP システムのメモリ管理にはガーベジコレクション（GC）が使用され，様々な手法があるが，TISL では次のような手法を使用している．

TISL のメモリ管理部では，メモリ領域を固定長領域と不定長領域の2つに分け，それぞれに対して独立した異なる GC 戦略を用いている．

図8 を用いて固定長領域の GC を説明する．固定長領域は CELL の集合で構成されており，初期化フェーズでは，live オブジェクトに灰色のマーキングをしておく．図8中の三角形は最後に割り当てられたセルを指している．メモリの割当てが要求されると空き領域に灰色のマーキングをしながらメモリを割り当てていく（図8(a)）．CELL を使い果たしたとき（図8(b)），固定長領域に対する GC が発生する．このとき，live オブジェクトに白のマーキングを行う．この後，メモリの割当てが要求されると空き領域に白のマーキングを行ってメモリを割り当てる（図8(c)）．すべてが白くマーキングされると（図8(d)），live オブジェクトに灰色のマーキングを行う．このようにして固定長領域の CELL の再利用を行っている．この方法では，メモリの割当て時に空き領域を探すために時間がかかることもあるが，GC 時にはスウィープやコピーを行う必要がなく，live オブジェクトへマーキングしか行わないため，中断時間が短くなる．

不定長領域に対しては**図9** に示すように，領域を2つに分割したコピー GC を使用している．不定長領域は必ず CELL から参照する形で使用しており，GC が発生したとき CELL へのマーキングを行い，live オブジェクトとなった CELL が不定長領域を使用しているとき，その領域をもう1つの領域にコピーすることにより GC を行う．この方式では GC 時に領域のコ

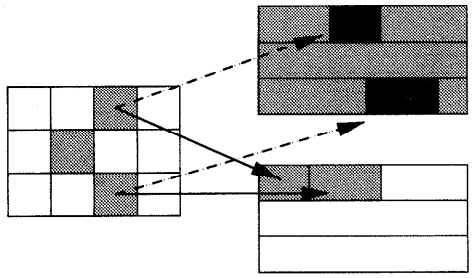


図9 不定長領域のGC
Fig. 9 GC for cells with variable length.

ビーが必要なため，固定長領域に対するGCよりも中断時間は長くなるが，不定長領域ではCELLとしては表現しにくい配列などの連続した任意の長さの領域を簡単に扱うことができる．

4.5 多重継承と包括関数の実現

ISLISPのクラス多重継承を実現するため，新しくクラスを作成するとき，指定された上位クラスのリストからクラス優先度リスト（2.2.1項参照）を作成する．同時に上位クラスのスロット記述をコピーし構造の継承を行う．同じ名前のスロットが存在した場合には，優先度の低い方がアクセス不能となる．下位クラスは上位クラスのすべてのスロット記述を持っていることになる．また，ここで作成したクラス優先度リストによって実効メソッドのメソッドの順番が決定される．

包括関数とは引数のクラスによって呼び出し時の振舞いが異なる関数である．ISLISPは多重継承を許しているので，TISLでは実引数のクラスのクラス優先度リストを参照しながら正しいメソッドが呼び出されるように実現されている☆．

包括関数が呼び出されると，最初に実引数のクラスの組合せが過去に存在したかどうかを包括関数が保持している表の中から調べる．表の中に存在する場合はこの実効メソッドを使用する．表の中に存在しない場合には，実引数のクラス優先度リストに従って適用可能なメソッドを並べて実効メソッドを作成し，実引数のクラスの組合せとともに表に登録する．

また，メソッドはcall-next-method局所関数を用いているか否かが中間コードを作成するときに検査されており，call-next-methodを用いている場合はメソッドの先頭の命令**MAKE-NEXT-METHOD**によってこの局所関数を作成する．call-next-methodを用いない場合にはこの局所関数を作成する必要はなく，局所関数作成のコストを削除できる．

このようにして実効メソッドおよびメソッドを作成し，メソッド組合せに従って，実効メソッドの中で一番優先度の高いメソッドを実行して包括関数の値を計算する．

以上のようにしてTISLインタプリタでは包括関数の効率の良い呼び出しを実現している．

5. TISLコンパイラ

TISLコンパイラではコンパイル時に指定された評価形式の実行のみを行う実行ファイルを作成する．この実行ファイルはインタプリタ上で実行するよりも高速に動作するアプリケーションとして利用することが可能となる．指定された評価形式の使用しているオブジェクトの環境初期化用の関数とその中間コードに対応する関数をC言語の関数として作成しファイルに出力する．このファイルをリーダ，ライタ，メモリ管理部などのTISLインタプリタと共通のファイルと一緒にC言語のコンパイラでコンパイルすることによって実行ファイルを作成する．コンパイル関数compileは第2引数に与えられた評価形式を実行するためのプログラムを第1引数で指定されたファイルに出力する．インタプリタのトップループで（compile “fact” (fact 10)）と入力し，実行の結果作成されるファイルを**図10**に示す．1行目は共通のインクルードファイル，2行目はこのプログラムファイル特有のインクルードファイルの指定である．3~5行目の関数Runはコンパイル時に指定された評価形式を実行する関数である．6~10行目の関数はコンパイル時に指定された評価形式を変換した関数であり，整数10をスタックに積んでfact関数を呼び出し，isTRUEを返して終了している．11~23行目はfact関数を変換して作成した関数である．24~32行目の関数InitializeObjectsは環境初期化用の関数であり，処理系の初期化の後に呼び出され，コンパイル時に指定された評価形式を実行するために必要なオブジェクト（この場合は2つの関数オブジェクト）の初期化を行う．

5.1 コンパイルされた関数の実行

TISLインタプリタの場合には，ISLISPで定義されている関数などのオブジェクトのすべての初期化や処理系の初期化を行ってから解釈実行のループに入る．TISLコンパイラの場合，処理系の初期化が行われた

☆ OpenLispでは多重継承をしたクラスに対して正しいメソッドを呼び出せない場合がある．6章の実験評価のgqsortプログラムに対してOpenLispは正しいメソッドを呼び出せなかった．OpenLispではメソッドをある優先度順で包括関数に登録しておき実行時に適用可能なものを（クラス継承関係を検査せずに）その順番で呼び出しているためと思われる．

```
1  # include "../include_ files.h"
2  # include "fact.h"
3  isBOOL Run() {
4      return cf7046488(0);
5  }
6  isBOOL cf7046488(isPCELL pEnvironment) {
7      cPushInteger(10);
8      cCallGlobalStack(cf7046044, 1);
9      return isTRUE;
10 }
11 isBOOL cf7046044(isPCELL pEnvironment) {
12     cNumberLessStackInteger(0, 2);
13     if (!ObjectNilP(SP-)) {
14         cPushInteger(1);
15     }
16     else {
17         cPushStack(-1);
18         cSubtractionStackInteger(-1, 1);
19         cCallGlobalStack(cf7046044, 1);
20         cMultiplication2();
21     }
22     return isTRUE;
23 }
24 isBOOL InitializeObjects(void) {
25     if (!cAllocateFunction(& function7046488,
26                            cf7046488, 0, 0))
27         return isFALSE;
28     if (!cAllocateFunction(& function7046044,
29                            cf7046044, 1, 1))
30         return isFALSE;
31     return isTRUE;
32 }
```

図 10 コンパイル例
Fig. 10 An example of compiled codes.

後，コンパイル時に指定された評価形式の使用しているオブジェクトの初期化関数（InitializeObjects）を呼び出し，初期化を行う．次に指定された評価形式の中間コードに対応する関数を呼び出す関数（Run）を実行し，値を計算する．

TISL インタプリタでは，4.3.5 項で述べたように，関数オブジェクトの中間コード配列に保持された C 言語関数へのポインタを順に呼び出して中間コードの実行を行う．これに対し，TISL コンパイラでは各命令に対応する C 言語のマクロが用意され，1 つの関数オブジェクトは基本的に 1 つの C 言語の関数として作成され，図 10 のように関数内に各命令がマクロで展開される．関数オブジェクトの実行はこの関数の呼び出しによって行われる．

5.2 中間コードの変換

各命令には，基本的に，対応する C 言語のマクロが用意されており，関数オブジェクトに対応する関数の中で順番に並べられる．

インタプリタでは大域関数の呼び出しは識別子経由で行っていたが，コンパイル時には関数は決定していなければならないので，直接，識別子を束縛している関数オブジェクトから作成される関数を呼び出す命令に変換する．各命令の動作はインタプリタとコンパイラで違いはないので局所変数の扱いはコンパイルしてもインタプリタの場合と同じである．メモリ管理や包括関数の呼び出しの機構はインタプリタにおける機構をそのまま使用している．

IF *then-form else-form*
(a) 命令 IF

```
if (!ObjectNilP(SP-)) {
        then-form の中間コードを展開したもの
} else {
        else-form の中間コードを展開したもの
}
```

(b) C 言語のプログラム

図 11 **IF** のコンパイル
Fig. 11 Compilation of **IF**.

なお，効率向上のため，TISL によるコンパイル時に C 言語にあわせて変換される中間コードも存在する．たとえば，図 11 のように，命令 **IF** は *then-form* と *else-form* の関数オブジェクトを引数にとるが，これらの関数オブジェクトに対応する C 言語の関数は作成されず，より直接的に C 言語の if 文を使用したプログラムへ変換される．

IF 以外に，**AND**，**OR**，**WHILE** などがこのような扱いをされ効率が改善されている．

5.3 型推論を用いた包括関数の効率良い実現

LISP コンパイラの効率の良い実現法として，古くから様々な最適化法[8]が提案されているが，現在の TISL コンパイラでは既述のような簡単な最適化（4.3.4 項参照）しか行っていない．しかし，6 章で報告するベンチマークプログラムでの結果によれば，TISL コンパイラは ISLISP の LISP 部分については十分に実用に耐える性能が得られていることが判明している．ISLISP の重要な特徴としてオブジェクト指向機能があるが，その実行効率を良いものとするため，TISL コンパイラでは，以下に述べるような，型推論を用いて包括関数の効率の良い実現を行っている．

包括関数の呼び出しの機構はインタプリタと共通の機構を使用している．しかし，TISL コンパイラでは，実効メソッドの決定に型推論を導入し，包括関数の実効メソッドが静的に決定できる場合には，直接実効メソッドを呼び出す命令を出力して包括関数の呼び出しの効率を改善している．

TISL コンパイラにおける現在の型推論の利用は，以下に述べるような簡便な方法である．

NUMBERLESS	引数の数 2
	第 1 引数 <number>
	第 2 引数 <number>
	返り値 <symbol>
PUSH-INTEGER	引数の数 0
	返り値 <integer>

(a) 命令の情報

PUSH-STACK *offset*
PUSH-INTEGER 2
NUMBERLESS

(b) 中間コード

図 12 型推論の例
Fig. 12 An example of type inference.

コンパイル時に型推論を行うので，型推論の対象は中間コードである．各命令に対して与えられているスタック上にとる値のクラスと返り値のクラスの情報からスタック上の値のクラスをトレースし，変数の属するクラスを推論していく．スタック上の変数を取り出す命令（**PUSH-STACK**）の場合はその変数の値を引数として使用する命令の情報から，以降でのその変数のクラスを推論できる．また，メソッドの中間コードを変換する場合には，メソッドの引数は特殊化されているため，あらかじめ引数に推論されたクラスを用意しておくことができる．

たとえば，図 **12** (a) のように各命令に対してスタック上にとる引数の数，各引数に期待されるクラス，返り値のクラスなどの情報が与えられている．**NUMBERLESS** は，引数を 2 つとり，両方とも <number> のインスタンスであると期待されており，返り値として <symbol> のインスタンスである t または nil をスタックに積む．**PUSH-INTEGER** はスタック上の引数をとらずにスタックに新しい整数（中間コードで与えられた値）を積む．

図 12 (b) のような中間コードの場合，**PUSH-STACK** と **PUSH-INTEGER** の返り値の情報から，**NUMBERLESS** の命令の直前ではスタック上にはこれまでに推論されている *offset* の位置の値のクラスと <integer> のインスタンスが積まれるはずである．**NUMBERLESS** は引数に <number> をとるのでこの中間コードが正常に終了するのであれば，この命令以後，スタック上の *offset* の位置の値は <number> のインスタンスであると推論できる．

スタック上のクラスをトレースしていき，包括関数の呼び出しの命令が現れたとき，スタックでトレースされたクラスが下位クラスを有しないクラスの場合，実効メソッドが決定できる．この時点で，実効メソッドを直接呼び出す命令を出力する．また，実効メソッドの中で最も優先度の高いメソッドが call-next-method を使用しないメソッドの場合にはメソッドの中間コードによる関数を直接呼び出す命令に変換する．これらの処理によって，実効メソッドを実行時に決定するコストや実効メソッドから関数オブジェクトを取り出すコストを削除することができる．

現在の方法では変数に対する推論はスタック上の局所変数に対しての静的推論のみ行っている．この場合，スタック上の局所変数が他の関数から参照されないことは中間コードに変換するときに保証されているので自分の中間コードを調べるだけでよい．

大域変数に対する型推論を行う場合には関数呼び出しのたびにその大域変数に対する副作用の有無を調べる必要があり処理が複雑になるので現在のシステムでは対応していない．また，包括関数の呼び出しの際，型推論されるクラスに下位クラスが存在する場合には実効メソッドを静的に決定することはできない．この場合には実行時にスタック上の値がこの下位クラスである可能性が存在し，下位クラスの状態によっては実効メソッドが変化するためである．このような動的な場合にはインタプリタの包括関数の呼び出しの機構を利用し実行時に実効メソッドを決定するようにしているのでインタプリタに比べて改善されていない．

型推論を行っているので必要のない型検査を中間コードから除く方式も考えられる．しかし，現在のシステムでは，型検査は 1 つの命令の中に埋め込まれており，型検査を行わない命令を新たに作成する必要がある．この場合，命令の数が膨大となってしまうので，そのような方式は採用していない．

インタプリタでは，1 つ 1 つの命令を大きくすることにより命令を読み込むための実行コストを小さくし，効率を上げている．コンパイル時にインタプリタで使用していた中間言語を型検査部と演算部などのように細かく分解し最適化を行いやすくする方法が考えられるが，それは今後の課題である．

6. 実験評価

現在，TISL は Windows および UNIX 上で動作している☆．**表 2** に Gabriel Benchmark プログラム[5]を PC（CPU AMD-K6 200 MHz, Memory 64 MB, OS WindowsNT4.0）上で実行したときの実行時間（単位は秒）を示す．表 2 から TISL インタプリタが OpenLisp よりも 1.3〜3.3 倍高速であり，TISL コンパイラ

☆ Windows は Microsoft Corporation のトレードマークであり，UNIX は X/Open Company Limited のトレードマークである．

表 2 Gabriel Benchmark プログラムによる結果
Table 2 Results of Gabriel Benchmarks.

(sec)

	OpenLisp インタプリタ	TISL インタプリタ	TISL コンパイラ	TISL/WS コンパイラ
tak	0.21	0.10	0.09	0.04
stak	0.29	0.19	0.13	0.10
ctak	0.37	0.19	0.16	0.10
takl	1.51	1.06	0.25	0.33
takr	0.25	0.16	0.16	0.07
boyer	-	1.95	0.59	0.68
browse	3.44	2.82	1.40	1.67
destructive	0.62	0.31	0.19	0.12
traverse-init	4.27	1.92	0.61	0.60
traverse-run	21.52	13.12	4.72	4.67
derivate	0.51	0.38	0.27	0.21
dderivate	0.55	0.40	0.27	0.24
div2-iter	0.45	0.29	0.17	0.17
div2-rec	0.42	0.27	0.17	0.16
FFT	2.62	0.79	0.35	0.25
puzzle	4.13	2.14	0.68	0.74
triangle	51.34	37.45	12.72	10.67

表 3 包括関数の実行例
Table 3 Examples of running several generic functions.

(sec)

	OpenLisp	TISL インタプリタ	TISL コンパイラ	型推論なし TISL コンパイラ
gfib	1.04	0.66	0.12	0.41
gtak	0.34	0.25	0.12	0.20
gderiv	1.71	1.08	0.83	0.92
gqsort	-	0.83	0.57	0.60
LispInLisp	1.35	0.71	0.40	0.40

が TISL インタプリタよりも 1.0~5.5 倍高速であることが知られる．なお，OpenLisp の場合 boyer ではリストのメモリが確保できず，実行ができなかった．

表 3 に PC 上での OpenLisp，TISL インタプリタ，TISL コンパイラおよび型推論なしの TISL コンパイラについて包括関数を使用したプログラムの実行例を示す．なお，gfib，gtak は引数を <integer> として作成した包括関数による fib，tak である．gderiv は包括関数を使用して微分関数を使い分けている derive である．gqsort はオブジェクトの比較に包括関数を使用して quick-sort を行うが，OpenLisp は多重継承をしているクラスに対して正しい実効メソッドが作成できず，正しい結果が得られない．gfib や gtak は包括関数の実効メソッドが静的に決定できる例である．この例から，包括関数について，TISL インタプリタは OpenLisp より性能が良いことが知られる．TISL コンパイラではさらに性能が改善されているが，特に静的に型推論が行える gfib と gtak の場合には，大幅に性能が改善されている．なお，TISL コンパイラから型推論を削除した型推論なし TISL コンパイラと TISL コンパイラの比較も行った．gderiv と gqosrt は静的に実効メソッドを決定できない例であり，実行時に実効メソッドを決定することになるため型推論の効果はない．また，インタプリタの場合と比較してもあまり性能向上がされていない．最後の LispInLisp の例は eval や apply を包括関数を使用して作成した LispInLisp 上で (fib 15) を計算したときの実行時間である．LispInLisp とは Lisp 言語を使用して Lisp 言語の処理系を書いたプログラムである．この例で使用している LispInLisp では用いられる関数が包括関数を使用して書かれている．メソッドを追加することにより容易に関数を追加できる例でもある．表 3 より，静的に実効メソッドが決定できない場合には，型推論が行えないため大きな効率の向上は望めない．しかし，型推論による効率の向上が望めないプログラムに対しても，TISL コンパイラは TISL インタプリタに比べ 2~3 割程度は効率が良くなっていることが知られる．これは，包括関数以外の関数についてはコンパイルによる効率が現れるためである．

なお，表 2 の TISL/WS コンパイラの欄は，ワークステーション Sun Enterprise3000 (CPU UltraSPARC-II 248 MHz × 2, Memory 512 MB, OS Solaris 2.5.1) 上での TISL コンパイラによるベンチマークプログラムの実行時間を示すものである☆．これによると Gabriel Benchmark プログラムの場合には，takl, boyer, browse, puzzle の場合は，PC 上での実行時間のほうが小さくなっている．tak, takr のように TISL/WS コンパイラでの実行時間が 0.10 秒以下の場合を除けば，PC 上での TISL コンパイラの実行時間とワークステーション上での TISL コンパイラの実行時間の差は 1.5 倍以内に収まっている．

表 4 にワークステーション Sun SPARC2 (CPU SUNW Sun4/75 75 MHz, Memory 32 MB, OS SunOS4.1.3) 上での KCL (Kyoto Common Lisp)[6] と TISL について Gabriel Benchmark プログラムを実行した結果を示す．インタプリタでは，KCL に比べて TISL が div2 の場合は 2~3 倍程度，その他の場合は 4~15 倍高速である．KCL コンパイラによる実行時間が 1 秒以内のものは，3~6 倍程度 TISL コンパイラが遅い．これは TISL コンパイラがインタプリタベースであるためインタプリタによる処理が相対

☆ Solaris は Sun Microsystems, Inc. のトレードマークであり，SPARC は SPARC International, Inc. のトレードマークである．

表4 KCLとの比較
Table 4 Results of comparing KCL and TISL.

(sec)

	KCL インタプリタ	KCL コンパイラ	TISL インタプリタ	TISL コンパイラ
tak	23.88	0.13	1.61	0.49
stack	20.34	0.42	4.14	1.21
ctak	26.00	1.47	4.22	1.24
takl	150.96	0.56	37.02	3.84
takr	28.77	0.17	1.91	0.80
boyer	265.47	4.24	48.05	7.53
browse	301.10	8.58	51.52	16.41
destructive	42.01	0.75	3.33	1.36
traverse-init	270.90	2.02	19.10	6.61
traverse-run	1673.88	10.67	122.70	50.85
derivate	27.20	2.63	6.90	2.34
dderivate	33.02	3.00	7.75	3.03
div2-iter	15.71	1.26	8.48	2.62
div2-rec	29.43	2.68	8.38	2.28
FFT	110.87	32.05	7.22	2.70
puzzle	329.83	1.18	22.63	7.66
triangle	4054.30	49.59	573.74	118.78

的に大きな比重を占めているためと思われる．KCLコンパイラによる実行が1秒以上の場合には，ctak, derivate, dderivateおよびdiv2-recのように，TISLコンパイラと差がないものもあるが，FFTの場合を除き，KCLコンパイラの方が2~7倍高速である．これは，KCLコンパイラの最適化がTISLコンパイラより充実していることによると考えられる．FFTについてTISLコンパイラはKCLの10倍以上高速になっており，このことから，TISLは浮動小数点数を効率良く処理できていることが確認できる☆．

TISLコンパイラは，インタプリタベースであり，また，最適化もあまり行われていないシステムとしては良い性能が出ていると考えるが，これは，ISLISPがコンパクトで実行効率の良い処理系が容易に作成できるようにとの視点から設計された言語であることに起因するとも考えられる[3]．

7. おわりに

TISLのインタプリタおよびコンパイラの概要とそれらの評価実験結果について報告した．ISLISPインタプリタとしてはOpenLispが発表されているが，TISLコンパイラはISLISPコンパイラとしては，筆者らの知る限り，初めての処理系である．また，TISLシステムは，Windows OSを備えたPC上での高速処理系となっている．

TISLは，KCLなどのCommon Lispの処理系と比較して，インタプリタベースではかなり高速に動作していることが確認できた．これはインタプリタがスタック形式の中間コードに変換し解釈実行していることと，ISLISPの仕様が非常にコンパクトであることによると考える．TISLコンパイラも，最適化の工夫をあまり行わずに作成したシステムとしては良い性能が出ていると考えるが，これもISLISPがコンパクトな言語であることに起因すると考える．TISLコンパイラにおける最適化機能の充実や型推論機能の充実などによって，さらなる性能の改善が期待できるが，今後の課題である．

TISLシステムに基づくエキスパートシステムや記号処理システムの実現などの応用も今後の課題である．また，ISLISPは，インターネット時代を指向したコンパクトなLISPという面も持っているが，そのためには，パッケージやモジュール機能の追加，JAVAバイトコードを出力するコンパイラの作成などが必要である．

参考文献

1) ISO/IEC 13816: 1997, Information technology – Programming languages, their environments and system software interfaces – Programming language ISLISP, p.126, ISO/IEC (1997).
2) JIS X 3012：1998プログラミング言語ISLISP, p.121, 日本規格協会 (1998).
3) 伊藤貴康：LISP言語国際標準化と日本の貢献，情報処理，Vol.38, No.10, pp.932–937 (1997).
4) Jullien, C.: OpenLisp, http://www.ilog.fr:8001/Eligis/ (1998).
5) Gabriel, R.P.: *Performance and Evaluation of Lisp Systems*, MIT Press (1985).
6) Yuasa, T.: Design and Implementation of Kyoto Common Lisp, *Journal of Information Processing*, Vol.13, No.3, pp.284–295 (1990).
7) Steele Jr., G.L.: *Common Lisp: The Language*, 2nd Edition, Digital Press (1990).
8) Allen, J.: *Anatomy of Lisp*, p.446, McGraw-Hill Company (1978).

付録

包括関数を用いたプログラムと型推論の例

包括関数を使用した例題で用いたgfibはfibの引数を <integer> と特殊化して作成したメソッドである．以下にfibとgfibのプログラムを示す．

☆ 比較に用いたKCLはオブジェクト指向機能をサポートしていないので，包括関数についての比較は行わなかった．

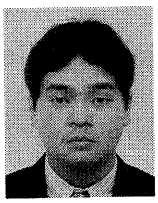
```
(defun fib (n)
  (if (< n 2)
      n
      (+ (fib (- n 1)) (fib (- n 2)))))

(defgeneric gfib (n)
  (:method ((n <integer>))
     (if (< n 2)
         n
         (+ (gfib (- n 1))
            (gfib (- n 2))))))
```

gfib がたとえば (gfib 25) のように呼び出された場合，最初に引数の 25 のクラス <integer> に対して適用可能なメソッドが包括関数 gfib の保持しているメソッドの中から選択される．浮動小数点数が引数として渡された場合には，<float> またはその上位クラスで特殊化されたメソッドがほかに定義されていなければ，適用可能なメソッドが存在せず例外が出力される．

TISL の場合は，過去に <integer> で呼び出されたときの実効メソッドが表に登録されているか否かが検査され，登録されていなければ，適用可能なメソッドを優先度順に並べた実効メソッドを作成する．もし，<number> で特殊化されたメソッドが存在した場合には，実引数のクラスが <integer> であるからクラス優先度リストのメソッドの順番は <integer>, <number> となる．実効メソッドが決定されると，その実効メソッドの中で最も優先度の高いメソッドが呼び出され包括関数の値を計算する．

TISL コンパイラで型推論を行った場合，gfib の再起呼び出し時の引数 (- n 1) と (- n 2) はメソッドの引数 n が <integer> と特殊化されていることとリテラル定数 1 または 2 との演算であることから結果の値は <integer> のインスタンスであることが推論できる．この結果，この 2 つの包括関数の再起呼び出しの実効メソッドが決定される．また，この場合，実効メソッドの最も優先度の高いメソッドが call-next-method 局所関数を使用していないことから，メソッドの本体部を変換して作成した関数を直接呼び出す命令に変換することができ，(gfib 25) の実行時間（0.12 秒）は包括関数を用いない (fib 25) の実行コスト（0.13 秒）より若干良いものとなっている．これは，fib で使用している中間コードでは引数のクラスが <integer> か <float> かを検査して実行時に動作を変化させる必要があるが，gfib の場合は引数のクラスが <integer> であることが保証されているので引数のクラスを検査する必要がなくなり，若干の効率改善につながっているものと考える．

(平成 10 年 11 月 30 日受付)
(平成 11 年 6 月 3 日採録)



泉　信人（学生会員）

1974 年生．1998 年東北大学工学部情報工学科卒業．同年東北大学大学院情報科学研究科情報基礎科学専攻博士前期課程進学．

伊藤　貴康（正会員）

1940 年生．1962 年京都大学工学部電気工学科卒業．スタンフォード大学コンピュータサイエンス学科および人工知能プロジェクト研究助手，三菱電機中研を経て 1978 年から東北大学．現職，東北大学情報科学研究科教授．工学博士．本会理事，東北支部長等を歴任．現在，Information and Computaion の Editor, Higher-Order and Symbolic Computation の Associate Editor, IFIP TC1 member 等．専攻分野，ソフトウェア基礎科学．日本ソフトウェア科学会，電子情報通信学会，人工知能学会，ACM 各会員．