LISP 上の GHC コンパイラ†

藤村 考†† 栗原正仁†† 加地郁夫††

並列論理型言語 GHC のサブセット Flat GHC の逐次処理系上での一コンパイル手法を提案し，ターゲット言語およびシステム記述言語に Lisp を用いてその処理系のインプリメントを行った．本コンパイラによって生成された Lisp プログラムはさらに Lisp コンパイラによってネイティブコードにコンパイルして実行できる．本方式の主な特徴は，①変数の管理およびスケジューリングが単純である，②Lisp を用いたため，コンパイラの保守や移植が容易である，ことにある．本方式は単純でありながらも，GHC の初期の応用に対してはある程度満足のいく効率を得ている．これは，①GHC のヘッドユニフィケーションはその受動的性質により，Lisp のバインディング機構により効率的に実現できる，②GHC にはバックトラック機能がないため，Prolog のような複雑な変数管理機構を必要としない，からである．また，幾つかの代表的なベンチマークテストにより，本コンパイル方法の有効性を示した．

1. はじめに

近年，計算機アーキテクチャの研究の成果により，従来のノイマン型アーキテクチャから，データフローモデルやリダクションモデルに基づく様々な高並列のアーキテクチャが提案されている．また，プログラミング言語の研究においても並列性をいかに自然に記述するかということが，大きな研究テーマとなっており，様々な並列プログラミング言語が提案，議論されてきた．並列論理型言語 GHC (Guarded Horn Clauses)[1] は，こうした中で生まれた並列性を陽に記述する言語であり，単純性，記述力，並列実行可能性等の点で優れた言語の一つである．現在，GHC を機械語とする並列マシンが研究，試作されているが，このようなハードウェアが広く使用されるようになるには，まだかなりの年月が必要と思われる．また，アプリケーションソフトウェアの開発はハードウェアの開発と共に進めることが望ましい．それは，アプリケーションの開発によって生じたハードウェアに対する要求をハードウェアの開発に反映させることができるからである．したがって，逐次計算機上で動作する実用的処理系が不可欠である．GHC の逐次処理系は DEC-10 Prolog 上のもの[2]をはじめとして既に幾つか発表されている．

本論文では，GHC を Lisp にコンパイルする一方法を提案し，その処理系の制作と評価を行った．本コンパイラによって生成された Lisp コードはさらに Lisp コンパイラによって機械語に変換して実行することができる．

Lisp をコンパイラのターゲット言語と記述言語とするのはあくまでも手段であって，本論文の内容は，他の言語をターゲット言語と記述言語としても有効である．著者らが特に Lisp を採用したのは以下の理由である．

① Lisp と GHC は共に記号処理言語であるため，セマンティックギャップが小さく，コンパイラの作成や保守が容易である．
② Lisp には Lisp 専用機を含め，優れた処理系が多数存在する．
③ Common Lisp による仕様統一化の動きにより，コンパイラの移植性が高い．
④ GHC におけるヘッドユニフィケーションは，その受動的性質[1]により，Lisp のバインディング機構によって容易にかつ効率的に実現できる．

からである．

本論文における Lisp プログラムおよび Lisp に関する用語は Common Lisp[9] に従っている．

2. 言語仕様

GHC は論理型言語 Prolog を基に発展してきた言語であり，論理 AND で結合された複数のリテラルのリゾリューションを並列に行うことを特長とする．プログラムは次のシンタックスをもつガード付きホーン節の集合で表す．

〈Head〉 :- 〈Guard Goals〉|〈Body Goals〉.

ここで，"|" はコミットオペレータと呼ばれ，〈Head〉

† An Implementation of GHC Compiler on the LISP Systems by KOU FUJIMURA, MASAHITO KURIHARA and IKUO KAJI (Department of Information Engineering, Faculty of Engineering, Hokkaido University).
†† 北海道大学工学部情報工学専攻系統工学講座

は一つのリテラル，〈Guard Goals〉，〈Body Goals〉はそれぞれ一つ以上のリテラルの並びである．また，コミットオペレータの左側（節のヘッドを含む）をガード，右側をボディと呼ぶ．

このようなガードを基本的同期機構に採用している並列論理型言語には GHC のほか，PARLOG[6]，Concurrent Prolog[7] 等があるが，GHC はそれらの言語と比較して，ガードの実行が受動的であるという特長をもつ．すなわち，ある節のガードの実行中に，その節を呼び出したゴールの変数にユニファイアを生成しようとすると，外から（他のゴールによって）ユニファイアが生成されるまで，実行が中断される．

本論文で対象とする言語は GHC のサブセット FGHC (Flat GHC[2),4)]) であり，GHC に以下の制限を加えたものである．

① ガードに書ける述語は組み込み述語のみ．
② ヘッドのユニフィケーションは left-to-right に実行する．
③ ガードのゴールは left-to-right に実行する．
④ 候補節のテストは上から下に実行する．

3. コンパイル方法の基本方針

本章では，GHC をインプリメントする際のポイントであるゴールのスケジューリング，サスペンション等について述べる．

GHC は論理 AND で結合されたゴールの展開を並列に実行する言語であるから，逐次計算機上で動作させるためには，何らかの方法で，ゴールをスケジューリングする必要がある．このスケジューリング技法として，上田らの処理系[3]と同様の bounded-depth-first を用いる．

N-bounded-depth-first とは，各ゴールが連続 N 回展開できることである．ゴール G が連続 N 回展開できるとは，G が展開されて，本体部のゴール $B_1, \cdots, B_m$ となったとき，$B_1, \cdots, B_m$ 以外のゴールに先んじて，各 B_i $(i=1, \cdots, m)$ が連続 $N-1$ 回展開できることである[3]．

このスケジューリングを実現するため，各ゴールを三つ組〈Pred, Bound, Args〉で表す．ここで，Pred はゴールの述語名 p およびアリティ r と一対一に対応する名前で，適当な関数 pred_arity (p, r) により一意に作られるものとする．たとえば，p="sort"，$r=3$ のとき，pred_arity (p, r) は "sort/3" を返す．Bound はそのゴールの連続展開可能回数であり，初期値は N で，一回展開されるごとに1ずつ減じられる．Args はゴールに与えられる引数の列である．ゴール G の三つ組の各要素はそれぞれ，Pred (G), Bound (G), Args (G) で参照されるものとする．このようなゴールの列を変数 Queue で表す．このとき，GHC 処理系のトップレベルを次のように構成する．

```
while (Queue≠空) do
  begin
    G := dequeue (Queue);
    resolve (G, Queue)
  end
```

ここで，dequeue は Queue の先頭のゴール G を値とする関数で，副作用として，Queue から G を取り除く．resolve はゴール G を最大 Bound (G) 回連続展開し，副作用として Queue の先頭や末尾に0個以上の新たなゴールを付け加える．

本論文では，与えられた GHC プログラム中の一つの述語定義（同一の述語名，アリティをヘッドに持つ節集合）ごとに一つの手続きにコンパイルすることを考える．この手続きは手続き名として，述語名とアリティから関数 pred_arity により定まる名前を持ち，仮引数として，連続展開可能回数 BD および述語の各引数に対応する $A_1, A_2, \cdots, A_r$ (r はアリティ）を持つ．すなわち，

```
procedure 述語名/アリティ (BD, A1, A2, ···, Ar);
  begin ··· end;
```

のような形で定義する．Queue はこの手続きに対して，大域的であるとする．ゴールの展開や Queue への副作用などすべての処理は，各述語定義ごとに作られたこれらの手続きで行う．このとき resolve が行う処理は，引数 Bound (G) と Args (G) に Pred (G) を適用することのみとなる．すなわち，前記の resolve $(G$, Queue$)$ は

```
apply (Pred (G), Bound (G). Args (G))
```

で置き換えられる．ただし，apply (F, S) は引数列 S に手続き F を適用する．$x.S$ は列の S の先頭に x を結合してできる列を表す．

各述語ごとに上記のような手続きがコンパイラにより生成される．この手続きの本体は，以下のようになる．

```
if BD=0 then {ゴールのスイッチ}
  "bound を N にリセットしたゴール 〈self, N,
  A1A2···Ar〉 を Queue の末尾に再スケジュール
  する."
```

else if "節 c_1 のガードが成功"
then "節 c_1 のボディの実行"
else if "節 c_2 のガードが成功"
then "節 c_2 のボディの実行"
else ……
else if "節 c_k のガードが成功"
then "節 c_k のボディの実行"
else {サスペンド}
"現在のゴール 〈self, BD, $A_1A_2\cdots A_r$〉 を Queue の末尾に再スケジュールする."

ただし，いまコンパイルしている述語定義は k 個の節 $c_1,\cdots,c_k$ から成っているものとする．self は上記の手続き自身の名前である．

節 c_i は次のような形をしているとする．

H :- $G_1, G_2, \cdots, G_m | B_1, B_2, \cdots, B_n$.

さらに，いま考えている述語名を p，アリティを r とし，H は $p(X_1, X_2, \cdots, X_r)$ の形をしているとする．

"節 c_i のガードが成功" の処理は "ヘッドユニフィケーション" と "ガードのテスト" の二段階から成っている．いずれの段階も，その実行結果は成功，失敗，中断のいずれかである．両段階とも成功のときに限り，節 c_i のガードは成功，すなわち条件 "節 c_i のガードが成功" は真となる．また，ここでは成功か否かのチェックが要求されているので失敗と中断の区別は必ずしも要求されていないことに注意．

"ヘッドユニフィケーション" はゴール G の引数 $A_1,\cdots,A_r$ とヘッド H の引数 $X_1,\cdots,X_r$ を単一化する試みである．(前述の処理系の構成により，この時点で，G と H の述語名とアリティは既に等しいはずなので，チェックする必要はない.) ただし，通常の定理証明手続きや Prolog とは異なり，GHC のヘッドユニフィケーションではゴール中の変数に対する代入は禁止されている．したがって，$A_i=X_i\theta$ $(i=1,2,\cdots,r)$ なる代入 θ が存在するときに限りヘッドユニフィケーションは成功する．これは並列処理において G を送信側，H を受信側と考え，G から H へのメッセージ送信による同期機構を実現するための制約である．上記の θ は存在しないが，$A_i\psi=X_i\psi(i=1,2,\cdots,r)$ なる代入 ψ が存在するとき，ヘッドユニフィケーションは中断，それ以外の場合は失敗となる．たとえば，ゴールが $p(X)$，ヘッドが $p([Y|Z])$ のときには，X がリストに具体化していない限りヘッドユニフィケーションは成功ではない．なおこの制約には各ガードを並列にテストする際に Concurrent Prolog のような多重環境[5]を作らなくてよいというインプリメント上の長所もある．

"ガードのテスト" は上記の代入 θ のもとでガードを実行することである．すなわち，$G_1\theta, G_2\theta, \cdots, G_m\theta$ が成功か否かチェックする (Flat GHC ではガードに書ける述語は組み込み述語のみであるためこのテストはただちに行える)．すべて成功のとき "ガードのテスト" は成功，少なくとも一つが失敗のとき "ガードのテスト" は失敗である．中断となるのはガード中の変数が適当なデータに具体化していないときである．例えば，数の大小比較を行うガード $X>Y$ において，$X=3, Y=1$ ならこのガードは成功，$X=1, Y=3$ なら失敗であるが，X と Y のいずれかが変数のままのときはガードは中断である．

"節 c_i のボディの実行" は，基本的には節 c_i のボディに対応した n 個のゴール $B_1\theta, B_2\theta, \cdots, B_n\theta$ を作り，Queue の先頭にスケジュールすることである．ただし，これらのゴールの Bound 値を現在の値 BD より，1だけ減じる．実際の処理系では効率上の理由のため，単一化 (=) や代入 (:=) などの直ちに実行できるゴール (個数を e とする) をただちに実行する．また，一般には，残りの $n-e$ 個のゴールのうち，$n-e-1$ 個は Queue の先頭にスケジュールされる．他の1個は直ちに実行 (展開) される．(すなわち，トップレベルには直ちに戻らず，このゴールに対応する手続きを直接呼び出す.)

サスペンドはいずれのガードも失敗または中断の場合の処理である．上の方式からわかるように，中断したゴールは単に Queue の末尾につながれ，動的待ち合わせ (busy-wait) を行う．なお，本来 GHC ではすべてのガードに失敗した場合はエラーとなるが，ここではこれをサスペンドに含めて扱っている．この主な理由は，すべてのガードに失敗することがあり得るか否かは静的にチェック可能な場合が多いことを考慮した上の効率上の理由である．また，いずれのゴールも無限に展開されることがないならば，いずれは上記のエラーとすべきゴールのみが Queue に残り，デッドロックとしてエラーを検出可能である．

本方式は，実際には，デッドロックの検出を行うため，もう少し複雑になる．デッドロックの検出は次のようにして行う．

Queue 内のゴール列中にダミーのゴール $end を一つだけ含めておく．$end は Queue から取り出される度に，単に Queue の末尾に移されるのみのゴール

である．$end が先頭に出現してからもう一度出現するまでに一つもゴールが展開されない場合にデッドロックと判定する．

4. コンパイル技法

本章では，第3章で述べた基本方針に基づいて，Lisp 上で効率的にインプリメントするための手法を具体的に述べる．

4.1 データ構造

データ型は以下の4種とする．

（1） 変　数
（2） 定　数
（3） 複合項
（4） リスト

ただし，(4)は概念上は(3)の一種と考えられるが，リストの効率的な処理のため，(3)と(4)を区別している．

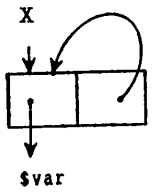
各データはいずれも大きさの等しい二つの領域〈tag〉および〈value〉から成るセルで表す．〈tag〉には(1)-(3)のデータを区別する特別なデータが格納される．ただし，(4)の場合には〈tag〉を通常のデータ領域として用いる．〈tag〉と〈value〉の大きさが等しいことが次節に述べるユニフィケーション法のための必要条件となる．これを Lisp で実現するには cons セルを用い，その car 部を〈tag〉，cdr 部を〈value〉とすればよい．具体的には，

（1） 変数は〈tag〉として特別のシンボル (distinguished symbol) $var をもち，〈value〉として単一化している変数セルへのポインタを持っている．単一化された変数は循環リストで結合されており，初期状態では，その変数セル自体を指している（図1）．

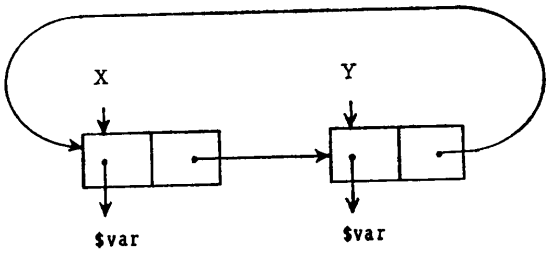
（2） 定数は〈tag〉として特別のシンボル $atom をもち，〈value〉としてアトムを持つ．

（3） 複合項は〈tag〉として特別のシンボル $rec をもち，〈value〉としてその複合項を表現するリスト(最初の要素は複合項のファンクタ，残りの要素は引数）へのポインタを持つ．

（4） リストは〈tag〉部，〈value〉部ともにデータ値を格納するのに用いる．〈tag〉部にはリストの第一要素，〈value〉部には第二要素以降のリストを格納する．したがって，これは Lisp におけるリストと全く同じである．このことが，リスト操作が効率的にできるポイントとなっている．



(a) A variable X.



(b) Unified two variables X and Y.

図 1 変数の構造
Fig. 1 Data structure of variables.

4.2 ユニフィケーション

GHC におけるヘッドユニフィケーション（ガードゴールとして，陽に記述する場合も含む）は，ボディに陽に記述するユニフィケーションとは違った意味である[1]．したがって，これらを区別して述べる．

（1） ヘッドユニフィケーション

第3章で述べたように，ヘッドユニフィケーションはゴール G の引数 $A_1, A_2, \cdots, A_r$ とヘッド H の引数 $X_1, X_2, \cdots, X_r$ を単一化する試みである．ただし，ゴール中の変数に対する代入は禁じられている．したがって，ヘッドユニフィケーションは $A_i = X_i\theta$ $(1 \leqq i \leqq r)$，を満たす代入 θ を求めようとする作業である．

この作業を第3章で述べたゴールの展開手続き（実際には一つの Lisp 関数）において

① Lisp のバインディング機構により，ゴール G の引数 A_i $(1 \leqq i \leqq r)$ を展開関数の引数として現れているシンボル（これを S_i $(1 \leqq i \leqq r)$ とする）にバインドする．

② 受け取った引数をセレクタ関数 (car, cdr 等) によって，必要な部分項をアクセスし，ヘッドユニファイ可能なように具体化されているかどうかテストする．

ことによって実現することとしよう．

上記の X_i はいまコンパイルしようとしている

GHC プログラム節のヘッドに引数として現れている項そのものであり，構造があらかじめ既知であるのに対し，A_i の具体的な構造はコンパイル時には一般には不明である．したがって，コンパイル時には具体的なθの値は求めることができない．その代わりに，X_i 中の各変数の位置をセレクタ関数を用いてコンパイル時に求めておく．②のテストを行うコンパイルコードはこの情報を用いて，引数 S_i とセレクタ関数および比較などの述語により，生成できる．(詳細は付録参照)

例：ヘッド $H=p([V|W], V, c)$ のとき，
各変数の位置情報は，
$\{car(S_1)/V,\ cdr(S_1)/W\}$．
②のコードは，
and (consp(S_1),
equal(car(S_1), S_2), equal(S_3, c))
となる．

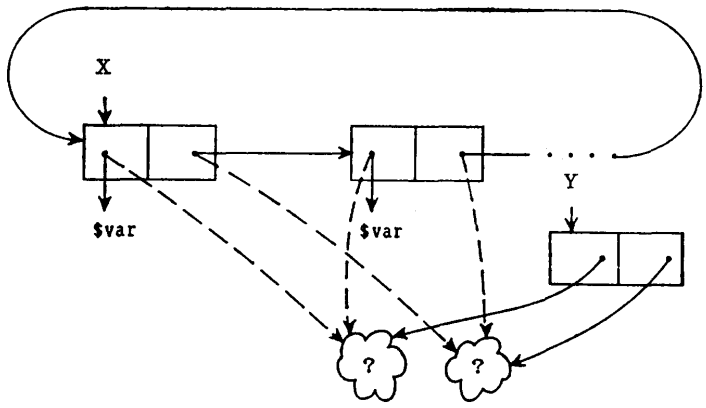
ここで得られた位置情報はガードおよびボディ中で変数を参照するのにも用いる．また，②のコードは第3章で述べたゴールの展開手続き中の“節 c_i のガードが成功”の部分の一部として使われる．

(2) ボディにおけるユニフィケーション

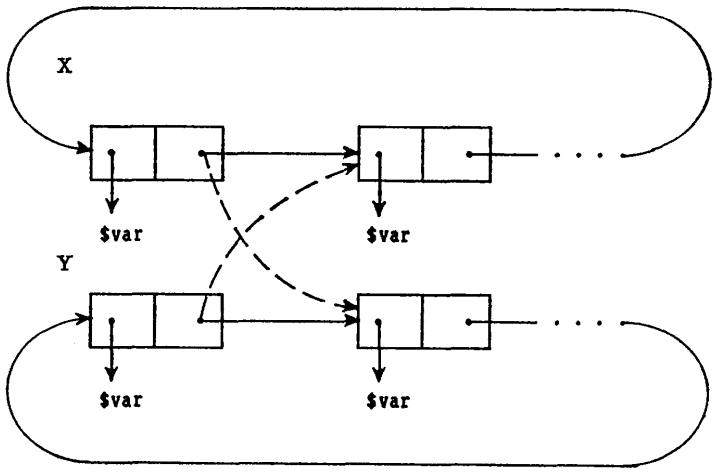
ボディのユニフィケーションはヘッドユニフィケーションとは異なり，変数に対する書き込みが行われる．本コンパイラはこれを副作用を用いて，変数セル自体を，直接，具体化した対象に書き換えることによって実現する．その主な理由は，

① ゴールの呼び出し側からその変数の具体化を観測できなくてはならない．一方，Lisp の変数のスコープは一般にその関数内である．
② デレファレンス (Dereference：ポインタをたどり変数の値を求めること) が基本的に不要になるため，高効率である．
③ メモリの使用効率が高く，不用セルの発生を抑えられる．
④ Prolog のようなバックトラックがないため，トレイルスタックを用いて，変数の束縛情報を残す必要がない．

からである．



(a) A variable X and a non-variable object Y.



(b) Two variables X and Y.

図 2 ユニフィケーション
Fig. 2 Unification.

変数セルを具体化した対象に書き換える様子を図 2 に示す．図2において，実線はユニフィケーション前のポインタの状態，破線はユニフィケーションによって書き換えられたポインタを示している．つまり，

(a) 変数が変数以外のオブジェクト (定数，複合項，リスト) に具体化した場合は，図2(a)のように，各変数セルをそのまま具体化したオブジェクトに書き換える．

(b) 変数同士のユニフィケーションをする場合は，図2(c)のように，循環リストを再構成する．

変数を循環リストで結んでいるのは，どの変数が先に非変数に具体化しても，他の変数セルをたどれるようにするためである．このユニフィケーションに要す

表 1 ガードにおける仮想命令
Table 1 Instructions in guard.

命　令	機　能
(IVAL X)	X の〈tag〉が $atom のとき X の〈value〉を返す*1)
(ARG X N)	X が複合項のとき，その第 N 引数を返す*1)
(INULL X)	X が空リストに具体化していれば T，それ以外は NIL を返す
(IEQUAL X Y)	X と Y が同一の基礎項に具体化していれば T，それ以外は NIL を返す
(IFUNC X F N)	X が関数記号 F アリティ N の複合項に具体化していれば T，それ以外は NIL を返す
(ICONSP X)	X が空でないリストに具体化していれば T，それ以外は NIL を返す
(GROUND X)	X が基礎項に具体化していれば T，それ以外は NIL を返す
(INEQUAL X Y)	X と Y が異なる項に具体化していれば T，それ以外は NIL を返す

*1) ボディの代入ゴールでも使用される．

る時間は循環リストの長さに比例する．しかし，GHC の実用的なプログラムでは，このような変数同士のユニフィケーションはほとんど行われず，長い循環リストが生成されることがまれなため，このことがユニフィケーションの効率の低下を導くことはない．

4.3 仮想命令とコンパイルコード

仮想命令は，COND 関数を初めとする幾つかの Lisp 関数をそのまま用いるが，新たに，**表 1, 2** に示すように，ガード 8 種，ボディ 8 種の命令を追加する．

ガードの命令は，第 3 章で述べた“節 c_i のガードが成功”の部分に生成されるコードに使用される．これらは IVAL と ARG を除いて，T か NIL を返す関数であり，副作用はない．例として INULL の定義を示す．

```
(defun INULL (X)
  (equal X ' ($atom . nil)))
```

IVAL は GHC のデータ構造を Lisp のデータ構造に変換するものである．この変換が必要となるのは，数式の評価が必要なガードゴールならびにボディの代入ゴールのみである．数式の評価をするのに，この変換が必要な理由は，次節の「Lisp とのリンク方法」で詳しく述べる．

ボディの命令は，“節 c_i のボディの実行”の部分に生成されるコードとして使用される．これらは基本的に副作用を使用しており，返す値は意味を持たない．すなわち，ADDQ と ENQ はグローバル変数である Queue を副作用的に書き換える．ALLOC はヘッドに現れていない変数（局所変数）が節 c_i にあるときにその変数セルを確保する．ND はグローバル変数であるデッドロックフラグを off にする．ユニファイ系の命令は，4.2 節で述べたように副作用的に変数セルを書き換える．

表 2 ボディにおける仮想命令
Table 2 Instructions in body.

命　令	機　能
(ADDQ G)	ゴール G を Queue の先頭に加える
(ENQ G)	ゴール G を Queue の末尾に加える
(ALLOC S)	変数セルを確保し，シンボル S に代入する
(ND)	デッドロックフラグを off にする
(UNIFY X Y)	X と Y を単一化する
(UNIL X)	X と NIL を単一化する
(UCONS X Y Z)	(CAR X) と Y, (CDR X) と Z を単一化する
(ASSIGN V X)	X を変数 V に代入する

コンパイルコード例として，数リストのクイック・ソートのプログラムのコンパイルコードを**図 3** に示す．QSORT/3 における $1, $2, $3 はそれぞれ局所変数 $S, L, Ys2$ に対応している．PART/4 で生成されたコードからわかるように，末尾再帰の除去を行い，スタックの消費を減らすことができる．

4.4 Lisp とのリンク方法

GHC の代入命令 $V:=E$ は数式 E を評価し，その結果と変数 V を単一化するというものであり，この命令は本質的に関数評価を行っている．したがって，この代入命令を拡張し，Lisp 関数を GHC の関数として使用することを許せば，新たなプリミティブを導入せずに最も自然な形で Lisp とリンクすることができる．また，この代入命令の右辺で使用される Lisp 関数を GHC 用に書き換える必要なく，そのまま使用できるようにすれば，システム関数（+, −, *, / 等）を一切作る必要がなくなり，処理系作成の手間を大幅に減らすことができる．このことは，ボディの代入ゴールだけではなく，数式の評価を行うガードゴールについても全く同様である．この場合問題となるのは，次の 2 点である．

```
% Example of quick sort.
% GHC source programs are as follows.
qsort([],Ys0,Ys3):-true | Ys3=Ys0.
qsort([X|Xs],Ys0,Ys3):-true | part(Xs,X,S,L),
    qsort(S,Ys0,[X|Ys2]),qsort(L,Ys2,Ys3).

part([],A,S,L):-true | L=[],S=[].
part([C|X],A,S,L):-A<C |
    L=[C|L1],part(X,A,S,L1).
part([C|X],A,S,L):-A>=C |
     S=[C|S1],part(X,A,S1,L).

% Compiled codes are as follows.
(DEFUN QSORT/3 (BD A B C)
  (COND ((ZEROP BD)
         (ENQ (LIST 'QSORT/3 *BD* A B C))
          ;*BD*には連続展開可能数の初期値Nが
          ;が代入されている.
        ((INULL A)
          (ND)
          (UNIFY C B))
        ((ICONSP A)
          (ALLOC '$3)
          (ALLOC '$2)
          (ALLOC '$1)
          (ND)
          (DECF BD)
          (ADDQ (LIST 'QSORT/3 BD $2 $3 C))
          (ADDQ (LIST 'QSORT/3 BD $1 B
                      (CONS (CAR A) $3)))
          (PART/4 BD (CDR A) (CAR A) $1 $2))
        (T (ENQ (LIST 'QSORT/3 BD A B C)))))
(DEFUN PART/4 (BD A B C D)
 (LOOP
  (COND ((ZEROP BD)
         (ENQ (LIST 'PART/4 *BD* A B C D))
         (RETURN))
        ((INULL A)
          (ND)
          (UNIL D)
          (UNIL C)
          (RETURN))
        ((AND (ICONSP A)
              (GROUND B)
              (GROUND (CAR A))
              (< (IVAL B) (IVAL (CAR A))))
          (ND)
          (UCONS D (CAR A) $1)
          (DECF BD)
          (PSETQ A (CDR A) D $1))
        ((AND (ICONSP A)
              (GROUND B)
              (GROUND (CAR A))
              (>= (IVAL B) (IVAL (CAR A))))
          (ALLOC '$1)
          (ND)
          (UCONS C (CAR A) $1)
          (DECF BD)
          (PSETQ A (CDR A) C $1))
        (T (ENQ (LIST 'PART/4 BD A B C D))
          (RETURN)))))
```

図 3 コンパイル例
Fig. 3 An example of compilation.

① 関数の評価を行う前に，その関数の引き数に現れている変数が基礎項に具体化しているかどうかテストする必要がある．そして，その結果，未代入変数があるときは，この代入ゴール自体をサスペンドしなければならない．

② GHC のデータ型を Lisp のデータ型に変換する必要がある．(〈tag〉 が $atom であるとき 〈value〉 を返す．)

そこで，本処理系では，例えば，次のような述語定義

$p(X, Y) :-$ true$|Y := X+1.$

をコンパイルした場合，以下のようなコードを生成するようにした．

```
(DEFUN P/2 (BD A B)
 (COND
   ((ZEROP BD) (ENQ (LIST 'P/2 *BD* A B)))
   (T (IF (GROUND A)
          (ASSIGN B (+ (IVAL A)1))
          (ENQ (LIST '$ASSIGN BD B (LIST A)
          #'(LAMBDA (A) (+ (IVAL A)1))))))))
```

このように，①の処理を行うために，ASSIGN を実行する前に，その引き数 A が基礎項に具体化しているかどうかテストし，もし，A が基礎項でなければ $ASSIGN という述語名のゴールがスケジュールされる．ただし，($ASSIGN BD X 変数リスト 関数) は変数リスト中のすべての変数が具体化されるのを待って，(ASSIGN X (APPLY 関数 変数リスト)) を行うゴールである．また，②の処理を行うために，代入ゴールの右辺の関数中に含まれるすべての変数に対して，IVAL 命令が加えられたコードを生成している．

これら①，②の処理により，Lisp のほとんどの関数をそのまま GHC の組み込み関数として使用可能になる．例えば，Lisp に sin という関数があれば，$Y := \sin(X)$ 等のゴールが使用できる．

また，代入ゴールの実行時には具体化されていることが保証される場合（ガードで既にチェックされている場合）は，代入ゴールの実行がサスペンドされることがないため，最適化したコードが生成される．例えば，述語定義

$P(X, Y) :- X>0|Y := X+1.$

は以下のようにコンパイルできる．

```
(DEFUN P/2 (BD A B)
  (COND
    ((ZEROP BD) (ENQ (LIST 'P/2 *BD* A B)))
    ((AND (GROUND A) (> (IVAL A) 0))
```

(ASSIGN *B* (+ (IVAL *A*) 1)))
(T (ENQ (LIST '*P*/2 *BD A B*)))))

5. インプリメンテーションとその評価

以上で述べたコンパイル技法に基づき，パーソナルコンピュータ PC-9801 上の muLISP 上，並びに大型汎用計算機日立 M-680 H 上の UTILISP 上で FGHC コンパイラを製作した．その結果，**表 3** に示すようなデータが得られ，本コンパイル技法の高速性が確かめられた．

ベンチマークプログラム Append-500 は 500 個の要素からなるリストのアペンドであり，Rev-30, Sort-50 は文献 8) に公表されている Prolog の代表的ベンチマークプログラムを GHC に直したものである．また，Bounded Buffer および Primes は GHC 独特の並列処理を行うものであり，このプログラムは文献 1) にある．表 3 において，(compiler) の列のデータは，本コンパイラによって Lisp にコンパイルされた GHC プログラムをさらに Lisp コンパイラでコンパイルした場合のデータであり，(interpreter) の列のデータは Lisp コンパイラを用いなかった場合のデータである．また，これらのデータはすべて，Bounded depth-first の連続展開可能数を 100 としたデータである．

測定は，Primes は 10 回，その他は 100 回繰り返して実行し，平均をとった．したがって，muLISP 上の処理系の場合，かなりの時間がゴミ集めに使われている．

Qsort-50, Bounded Buffer (size=10), Primes, の RPS (Reduction Per Second) 値が Append-500, Rev-30 の値の 1/2 以下になっているのは，ガードゴールとボディの単一化と代入ゴールの実行を Reduction としてカウントしていないためである．したがって，LIPS (Logical Inference Per Second) 値ではいずれのプログラムも 200 [kLIPS] (M 680) 前後である．

PC-9801 上の処理系は M 680 上のものと比較して，インタプリタで 1/20, コンパイラで 1/100 程度の速度低下であった．これは Lisp 自体の速度の差とほとんど同じである．また，文献 8) に公表されている Lisp のベンチマークプログラム [2-4], [2-5] による Lisp 自体の最高速のコードと比較した場合の速度低下は，コンパイラで，1/4.5〜1/5.0，インタプリタで 1/2.5〜1/5.0 程度であった (**表 4**)．これは Prolog による GHC 処理系のオーバヘッドと同程度である．

本処理系は現時点では，モード宣言および bounded-depth-first 以外のスケジューリングはサポートしていない．また，仮想命令はいずれも Lisp の関数定義 (defun) またはマクロ (defmacro) で定義されており，マシンコードやマイクロコード化していない．したがって，今後，それらの改良を行うことによって，さらに高速化することができる．

表 3 処理能力の評価
Table 3 GHC benchmarks.

GHC Programs	Machine	Reductions	Suspensions	Time[ms]/KRPS*1)	
				(compiler)	(interpreter)
Append-500	PC*2)	502	0	310/1.62	560/0.90
	M 680*3)	502	0	2.23/225	35.9/14.0
Rev-30	PC	497	0	280/1.78	560/0.89
	M 680	497	0	2.35/211	35.4/14.0
Sort-50	PC	377	0	420/0.89	700/0.54
	M 680	377	0	3.97/95.0	34.4/10.9
Bounded Buffer (size=1)	PC	205	200	420/0.49	720/0.28
	M 680	205	200	5.53/37.1	24.5/8.36
Bounded Buffer (size=10)	PC	214	20	280/0.76	440/0.48
	M 680	214	20	2.45/87.3	19.4/11.1
Primes (2 to 300)	PC	2,715	73	4,900/0.55	7,200/0.38
	M 680	2,715	73	50.5/53.8	283/9.59

*1) number of reductions per second.
*2) muLisp-86 on NEC PC-9801 VM (10 MHz, 384 kbyte Memory).
*3) UTILISP on HITAC M-680 H, VOS 3.

表 4 Lisp の速度との比較
Table 4 Comparison with Lisp.

Lisp Programs (Ref. 8))	Machine	Time[ms]/Ratios ((Time of Lisp)/(Time of GHC))	
		(Compiler)	(Interpreter)
Append-500	PC	65/0.20	240/0.43
	M 680	0.50/0.22	6.96/0.19
Rev-30 ([2-5])	PC	60/0.21	220/0.39
	M 680	0.51/0.22	6.68/0.19
Sort-50 ([2-4])	PC	85/0.20	300/0.43
	M 680	0.55/0.14	8.13/0.24

プログラムの大きさは，仮想命令の定義が約200行，コード生成部が約 300 行，DEC-10 Prolog 準拠のシンタックスをS式に変換するパーザが約300行，組み込み述語約 50 個の定義が約 400 行，その他が約 200 行であり，計 1,400行程度で記述することができた．これは GHC と Lisp とのセマンティックギャップが小さいことを示している．

プログラミング環境については，本処理系はコンパイラでありながら，コンパイルはプログラムの reconsult 時に行うため，ユーザからは Lisp を中間言語とするインタプリタ，あるいは，Lisp のプリプロセッサのようにみえる．それゆえ，プログラムの開発，デバッグをインタプリタのような手軽さで行えるという利点があることも確かめられた．また，4.4 節で述べたように，代入命令の右辺に Lisp 関数を GHC の組み込み関数のように，そのまま使用できるため，Lisp 関数を利用した本格的なアプリケーションを GHC の枠組みの中で作ることができる．

6. むすび

並列論理型言語 GHC の逐次処理系上での一コンパイル手法を提案し，ターゲット言語およびシステム記述言語として Lisp を用いて，その処理系のインプリメントを行った．また，ベンチマークテストによる性能評価を行い，本処理系の高速性を確認した．

本処理系の特徴の一つは，既に述べたように，単純で，その保守や拡張が容易である点にある．これは，GHC の仕様の細部が今後微妙に変わり得ること，およびスケジューリング法の改善やプログラミング環境の整備などの将来の課題を見越している．

また，本処理系は単純でありながらも，GHC の初期の応用に対しては，ある程度満足のいく効率を達成している．汎用機上で，より効率的な処理系を得るためにはC言語などで本格的に開発する必要があるが，本処理系はその際の有力なプロトタイプを提供しているともいえる．

謝辞 プログラミング言語 GHC を提案した上田和紀氏，および，有益な助言をしていただいた，久野英治氏に深謝する．

参考文献

1) Ueda, K.: GUARDED HORN CLAUSES, TR-103, ICOT (1985).
2) 上田，竹内：GHC プログラミング講習会資料，ICOT (1986).
3) 上田，近山：並列論理型言語の実用処理系，日本ソフトウェア科学会第 1 回論文集，pp. 307-310 (1984).
4) 江崎令子ほか：GHC のサブセット逐次処理系の作成，第 32 回情報処理学会全国大会講演論文集，2G-4 (1986).
5) 宮崎敏彦，佐藤裕幸，田中二郎ほか：Concurrent Prolog のシーケンシャルインプリメンテーション，日本ソフトウェア科学会第 1 回論文集，pp. 295-306 (1984).
6) Clark, K. L. and Gregory, S.: PARLOG: Parallel Programming in Logic, Research Report DOC 84/4, Dept. of Computing, Imperial College, London (1984).
7) Shapiro, E. Y.: A Subset of Concurrent Prolog and Its Interpreter, TR-003, ICOT (1983).
8) 奥野 博：第3回 LISP コンテスト及び第1回 Prolog コンテストの課題案，記号処理，28-4 (1984).
9) Steele, G. L., Jr.: *Common LISP*, Digital Press (1984). (後藤英一監訳，井田昌之訳：Common LISP，共立出版，*bit* 別冊 (1985)).

付録 ヘッドユニフィケーションのテストを行うコンパイルコードの生成法

展開関数の引数列を $S=S_1S_2\cdots S_r$，ヘッドの引数列を $X=X_1X_2\cdots X_r$ とする．

準備として，アクセス式（以下，「A 式」と略）という特別な項を定義しておく．A 式とは，S に渡される構造の任意の部分項にアクセスするときに評価される式を表す項であり，構文的には次のいずれかの形をしたものである．

S_i：第 i 引数にアクセスする．

$arg ($A$ 式, n)：A 式でアクセスされる複合項の第 n 引数．

$car ($A$ 式)：A 式でアクセスされるリストの car 部．

$cdr ($A$ 式) : A 式でアクセスされるリストの cdr 部.

項の列 $E=E_1E_2\cdots E_q$ と $T=T_1T_2\cdots T_q$ および代入 σ が与えられたとき，$E=(T\sigma)\lambda$ なる代入 λ の存在をテストする論理式を生成する関数を test(E, T, σ) とする．test の返す値は論理式 F と代入 θ のペア $\langle F,\theta\rangle$ である．θ は上記の λ が存在するときの代入であり，$\theta=\sigma\circ\lambda$ (σ と λ の合成) で与えられる．すなわち，F が真ならば $E=T\theta$ が成立する．したがって，トップレベルで，test(S, X, ϕ) (ϕ は空代入) を求めれば目的のコードが得られる．

論理式 F は基本論理式の連言として与えられる．基本論理式は次の 4 種類のいずれかの形とする．

equal (A 式, 基礎項)
equal (A 式$_1$, A 式$_2$)
functor (A 式, 関数記号, 整数)
consp (A 式)

基礎項とは変数を一つも含まない項である．functor (A 式, f, n) は A 式でアクセスされる項が，主関数記号 f および n 個の引数からなる複合項であるときに限り真となる．consp (A 式) は A 式でアクセスされる項が空でないリストのときに限り真となる．

便宜上，$F_1\wedge\langle F_2,\theta\rangle=\langle F_1\wedge F_2,\theta\rangle$ と定義すると，test は次式で定義される．ただし，ε は空列を表す．また，$E_1.E'$ は項 E_1 と項の列 E' を結合してできる列を表す．

test(ε, ε, σ)=$\langle$true, $\sigma\rangle$
test($E_1.E'$, $T_1.T'$, σ)=

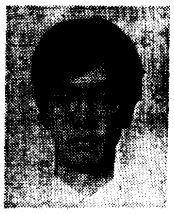
① $T_1\sigma$ が変数 V のとき，
test(E', T', $\sigma\circ\{E_1/V\}$)

② $T_1\sigma$ が基礎項または A 式のとき，
equal(E_1, $T_1\sigma$)$\wedge$test(E', T', σ)

③ $T_1\sigma$ が②以外の複合項 $f(Y_1,\cdots,Y_s)$ のとき，
functor(E_1, f, s)$\wedge F_1\wedge$test(E', T', σ_1)
ただし，$\langle F_1,\sigma_1\rangle$=
test($arg($E_1$, 1)$\cdots$$arg(E_1, s), $Y_1\cdots Y_s$, σ)

④ $T_1\sigma$ が空でないリスト $[Y_1|Y_2]$ のとき，
consp(E_1)$\wedge F_2\wedge$test(E', T', σ_2)
ただし，$\langle F_2,\sigma_2\rangle$=
test($car($E_1$)$cdr(E_1), Y_1Y_2, σ)

(昭和 61 年 11 月 25 日受付)
(昭和 62 年 5 月 13 日採録)



藤村 考 (正会員)

昭和 36 年生．昭和 59 年北海道大学工学部電気工学科卒業．昭和 61 年同大学院修士課程修了．現在，同情報工学科博士課程在学中．論理型プログラミングおよび協調計算アルゴリズムに興味を持つ．日本ソフトウェア科学会会員．

栗原 正仁 (正会員)

昭和 30 年生．昭和 55 年北海道大学大学院工学研究科情報工学専攻修士課程修了，同年同大学工学部助手．現在，同情報工学科助手．工学博士．システム工学，知識工学の研究に従事．電子情報通信学会，電気学会．日本 OR 学会，日本シミュレーション学会，日本ソフトウェア科学会，IEEE 各会員．

加地 郁夫 (正会員)

大正 15 年生．昭和 26 年北海道大学理学部数学科卒業．同年北海道大学応用電気研究所助手，40 年北海道大学工学部助教授，44 年同教授，現在，同情報工学科システム工学講座教授．システム工学，応用数学の研究に従事．日本 OR 学会，日本物理学会，電子情報通信学会，計測自動制御学会，日本原子力学会各会員．